Dérivation de programmes impératifs à partir de spécifications algébriques

Yves Guerte

N^o d'ordre : 266-96 Année 1996

THÈSE

présentée

devant l'UNIVERSITÉ CLAUDE BERNARD – LYON I –

pour l'obtention

du DIPLÔME DE DOCTORAT

(Arrêté ministériel du 30 mars 1992)

Spécialité : Informatique

par

Yves GUERTE

soutenue le 28 octobre 1996

Dérivation de programmes impératifs à partir de spécifications algébriques

---

Composition du jury :

Président :	M. Jean-Marc FOUET
Rapporteurs :	Mme. Christine CHOPPY
	M. Jean-Pierre FINANCE
Examinateur :	M. Didier BERT

Préparée au Laboratoire Logiciels Systèmes et Réseaux (LSR–IMAG) de Grenoble

Remerciements

Je remercie les membres de mon jury, et tout d'abord mon directeur de thèse, M. Didier Bert, pour la confiance, l'attention et la disponibilité qu'il m'a accordées durant la préparation de cette thèse. Je remercie ensuite:

- M. Jean-Marc Fouet pour avoir accepté de présider ce jury (ainsi que pour la dynamique et l'intérêt qu'il a su insuffler à son cours de DEA) ;
- Mme. Christine Choppy pour le soin qu'elle a pris à rapporter sur ma thèse, et pour la gentillesse avec laquelle elle m'a donné ses remarques ;
- M. Jean-Pierre Finance pour avoir accepté d'être rapporteur de ma thèse, pour la pertinence de ses remarques, et le recul qu'il m'a permis de prendre sur mon sujet.

Je remercie les membres de l'équipe SCOP pour leur accueil et les conseils qu'ils ont pu me donner.

Je suis très reconnaissant envers Catherine Oriat et Pascal Brand pour les coups de main qu'ils m'ont donnés au long de cette thèse ou pendant les moments du sprint final. Un grand merci également à Hervé Raynaud et aux "stagiaires du Sappey" pour leur "mise en forme psychologique".

Je remercie ma famille, et surtout ma femme, Sonia, pour son soutien, l'énergie, la motivation et les encouragements qu'elle m'a prodigués à tous les instants.

Résumé

Ce mémoire présente une méthode de dérivation automatique des spécifications algébriques vers un langage impératif. Par langage impératif nous désignons un langage de programmation "traditionnel" avec déclarations des variables (état) et dont les programmes sont des suites d'instructions qui modifient l'état. L'instruction caractéristique est l'affectation destructrice d'une valeur à une variable. Une spécification algébrique est composée de sortes, de constructeurs qui définissent l'ensemble des valeurs atteignables (que l'on peut dénoter), et d'opérateurs axiomatisés par des équations conditionnelles orientées.

Nous définissons un lien d'implémentation entre les objets du domaine abstrait des spécifications algébriques et ceux du domaine concret des programmes impératifs. Ce lien permet de paramétrer la dérivation d'une spécification. L'implémentation des opérateurs respecte le choix de la forme de dérivation et celui de la bibliothèque importée. Elle résout les conflits d'accès aux variables et minimise les coûts en recopies de valeurs engendrées par le passage du fonctionnel à l'impératif.

De manière analogue au lien d'implémentation entre une spécification algébrique et un programme impératif, nous définissons un lien d'implémentation dite abstraite entre les sortes et constructeurs de deux spécifications algébriques. Nous proposons pour les constructeurs, soit d'effectuer une dérivation systématique en un type de donnée impératif, soit de calculer les liens d'implémentation abstraite potentiels vers les sortes dont les implémentations des constructeurs sont réutilisables.

Une méthode de transformation de la spécification algébrique est enfin proposée, qui favorise les modifications "en-place" de données, dans une variante de la méthode de dérivation précédente.

Mots-clés : Spécifications algébriques, dérivation de programmes, réutilisation d'implémentation, optimisation d'implémentation, transformation fonctionnel-impératif.

Abstract

This thesis presents a method to automatically derive programs in an imperative language from algebraic specifications.

The imperative language is a standard one i.e. programs contain declarations of variables (the state) and sequence of statements which modify the program state. A characteristic feature is the destructive assignment of a value to a variable. An algebraic specification is composed of sorts, constructors which define the set of the possible values (which can be denoted), and operators which are axiomatized using conditional rewrite rules.

An implementation link is defined between the objects of the algebraic specification domain, called the abstract domain, and the imperative program domain, called the concrete domain. The formalization of this implementation link allows one to parametrize the derivation of a specification. The implementation of the operators respects the form of the derivation and the contraints of the imported library. It also solves the access conflicts to the variables, and minimizes the cost involved by the duplications generated by the transformation from the functional to the imperative style.

Similarly, we define an implementation link, called abstract link. It maps sorts to sorts and constructors to constructors from an algebraic specifications to another one. We propose two kinds of derivation for the constructors: either we use a systematic derivation to an imperative data type, or we compute all the possible abstract implementation links to the sorts whose implementations can be re-used.

Finally, a method which enhances the partial modifications of the data is proposed. It relies upon the same principles as the ones developed previously.

Keywords : Algebraic specifications, program derivation, implementation reuse, optimisation of implementation, fonctional-to-imperative transformations.

Table des matières

Table des matières **7**

Introduction **9**

1 Spécifications formelles et programmes **15**
1.1 Les spécifications algébriques 15
1.1.1 Concepts syntaxiques de base 15
1.1.2 Sémantique 17
1.1.3 Notre langage de spécification algébrique 18
1.2 Les programmes impératifs 20

2 Des spécifications formelles aux programmes **25**
2.1 Développements 26
2.1.1 L'implémentation abstraite 26
2.1.2 Le raffinement 27
2.1.3 Larch : liens spécifications – programmes par preuves 28
2.1.4 Techniques par "plans de travail" 29
2.1.5 Synthèse 30
2.2 Dérivations 31
2.2.1 Transformations 31
2.2.2 Dérivations du Fonctionnel vers l'Impératif 31

3 Méthode de dérivation vers des programmes **41**
3.1 Cadre et problématique 41
3.1.1 Système favorisant la réutilisation 41
3.1.2 Lien d'implémentation 44
3.1.3 Problématique liée à l'introduction de procédures 48
3.1.4 Spécification source 49
3.1.5 Formalisme 53
3.2 Principe de la dérivation 60
3.2.1 De la signature algébrique à l'interface impérative 60
3.2.2 Mise en œuvre du lien Δ dans la dérivation par $\Delta^{\#}$ 63

4 Résolution des conflits d'accès aux variables **71**
4.1 Motivations, notations, principe 71
4.1.1 Variable = valeur référencée par un nom 71
4.1.2 Localisation des conflits 73

4.1.3 Sous-ensembles d'accès conflictuels indépendants 75
4.1.4 Raisons et conséquences du choix d'un ordre d'évaluation entre plusieurs sous-termes 76
4.1.5 Arbre abstrait 77
4.1.6 Principe 79
4.2 Calcul des destinations 85
4.3 Calcul des relations $\mathcal{R}_<$ 90
4.4 Calcul des cycles dans les relations $\mathcal{R}_<$ et $\mathcal{R}_T$ 95
4.5 Calcul des coupures des cycles 96
4.6 Application des coupures de cycles 99
4.7 Traduction du graphe d'évaluation en code impératif 102

5 Compilation des constructeurs **109**
5.1 Dérivation systématique 109
5.2 Implémentation par un type existant 112
5.2.1 La bibliothèque 112
5.2.2 Réutilisation de l'implémentation des constructeurs 113
5.2.3 Perspective : réutilisations partielles 117

6 Flots et mutations **121**
6.1 Dérivation avec destinations des sous-termes fixées 121
6.1.1 Calcul de la destination du résultat de chaque sous-terme 124
6.1.2 Calcul des conflits d'accès aux variables 126
6.1.3 Minimisation optimale des duplications 127
6.2 Mutations 129

7 Outil **131**
7.1 Description de l'outil 131
7.2 Syntaxe du lien d'implémentation 131
7.3 Exemples – résultats 135

Conclusion **141**
Bilan 141
Perspectives 143

Références bibliographiques **147**

Introduction

Les spécifications formelles

L'informatique atteint quasiment tous les domaines, automatise la résolution des problèmes les plus divers et les tâches les plus quelconques. Les développements informatiques doivent donc s'adapter à un plus grand nombre de contraintes :

- le domaine d'application est très pointu, très spécialisé : la description d'un problème doit être faite sans ambiguïté entre le spécialiste client et l'informaticien,
- le domaine d'application est à risques : les résultats doivent être garantis corrects dans toutes sortes de conditions d'utilisation (délais, précision, récupération d'erreurs humaines, etc),
- le domaine d'application évolue : la solution informatique doit faire de même sans être du coût d'une refonte complète, même si la maintenance est assurée par des informaticiens externes à la conception initiale.

De plus, le client doit avoir une vue précise de l'application, sans en attendre la fin de la réalisation.

Toutes ces contraintes font qu'il est nécessaire à l'informaticien ou au groupe d'informaticiens, d'avoir à sa disposition des méthodes et des outils. Les spécifications formelles, sans être la panacée, car on leur reproche souvent leur lourdeur, et le fait de n'être pas toujours comprises par des personnes autres que des spécialistes, ont de nombreux avantages :

- la non ambiguïté et la clarté de leur langage en séparant la syntaxe du contenu exprimé : la sémantique. La réutilisabilité est simplifiée et la correction de la composition avec des spécifications existantes peut être vérifiée formellement.
- l'apport d'un prototype du modèle qu'elles spécifient, si la spécification est exécutable.
- la possibilité d'être à l'origine d'une réalisation informatique (un programme) sans orienter cette réalisation. Un développement ou une dérivation vers un programme est alors envisageable d'une manière formelle, la classe d'outils ou de problèmes spécifiés étant décrite dans un langage avec une sémantique précise.
- la validation et la vérification de leurs propriétés : la construction de la spécification est-elle correcte, cohérente, complète. La dérivation ou le développement à partir des spécifications permet également de tenir compte du comportement désiré pour le programme final obtenu par ajout de nouvelles informations.

J.P. Finance, dans [Fin79] décrit fort bien les motivations qui conduisent à spécifier et les différentes qualités des spécifications formelles : existence de méthodes de construction, largeur du domaine d'application, facilité de compréhension (lisibilité), de vérification, de résolution ultérieure, primitives de haut niveau, support graphique, etc.

Parmi les différents formalismes existants, réseau d'automates, graphes conceptuels, machines abstraites, spécifications logiques, spécifications algébriques, nous nous sommes particulièrement intéressés à ce dernier. Les types abstraits algébriques dont les premières références sont [Gut75], [Gut77] ont fait depuis l'objet de nombreuses recherches théoriques.

Évolutions d'une spécification formelle

La spécification formelle peut être un but en elle-même si elle ne sert que de "cahier des charges" très précis pour l'informaticien, qui développera ensuite un programme implémentant la spécification. Mais dans la plupart des cas, la spécification formelle sert de source à un développement, une dérivation :

- soit le développement fait partie du processus de spécification : la spécification source est extrêmement succincte (une simple signature par exemple) et le développement consiste à ajouter des caractéristiques des spécifications : choix de méthodes de décomposition et de re-composition d'objets par des schémas d'induction, des alternatives, des applications de plan de travail [Los91],[Sou93],

- soit les spécifications ne sont pas exécutables : il est souhaitable de les transformer en un formalisme où nous pouvons plus facilement les exécuter, calculer leurs propriétés, ou les implémenter par des programmes existants. Un avantage à privilégier le travail fait en amont de l'implémentation, c'est-à-dire au niveau de la spécification est que, contrairement à l'implémentation qui peut utiliser des structures de données et des invariants complexes pour engendrer l'efficacité, la spécification ne décrit que les données nécessaires et suffisantes à la définition d'un problème. Il s'ensuit des modifications plus faciles au niveau de la spécification qu'au niveau du code (éventuellement optimisé par la dérivation).

Notre approche

Le problème auquel nous nous attaquons est le passage du paradigme fonctionnel au paradigme impératif. Le premier ne manipule que des valeurs, alors que le second se décrit par des états définis par un environnement. Les théories mathématiques sur lesquelles sont basées ces deux formalismes sont :

- les algèbres, le λ-calcul et les combinateurs, qui permettent de décrire la sémantique des programmes fonctionnels,

- la logique de Floyd et Hoare [Hoa69] : un système formel qui permet de prouver la correction des programmes impératifs.

Nous proposons une méthode de dérivation automatique d'un type algébrique (sorte, constructeurs et opérateurs) ou enrichissement (opérateurs seuls) vers un paquetage

impératif (type, fonctions et/ou procédures). Nous formalisons le lien entre les objets du domaine abstrait (spécifications algébriques), et ceux du domaine concret (programmes impératifs).

Nous donnons le lien d'implémentation d'une spécification algébrique pour préciser le résultat que nous voulons obtenir par la dérivation. Ce lien d'implémentation "paramètre" en quelque sorte la dérivation d'une spécification algébrique, et permet également de réutiliser un programme impératif dès lors que nous savons lui associer une spécification via un lien d'implémentation. Ce programme sera alors utilisé par tout code dérivé d'une spécification qui importe la spécification du programme.

Les avantages que nous apportons par notre méthode sont les suivants :

Gestion de la mémoire plus efficace : La solution adoptée par la majorité des langages de spécifications algébriques ou des langages fonctionnels, est une gestion automatique de la mémoire par l'intégration des techniques de ramasse-miettes. Ce handicap est à l'origine de nombreux travaux pour supprimer ces techniques, par des analyses statiques par interprétation abstraite, ou par optimisation de l'algorithme de ramasse-miette et de la représentation des données (dans [Sig95] sont notés des exemples de 70 à 90% de mémoire allouée et non référencée en fin d'exécution, ainsi que des copies multiples des objets qui représentent 50% du total de la mémoire allouée). Nous proposons donc de transformer des spécifications algébriques (exécutables car l'axiomatisation des opérateurs est définie par des équations orientées) en des programmes impératifs. Nous introduisons des modifications "en place" de données. Les langages impératifs permettent en effet l'affectation destructive d'une variable et l'appel de procédure. Nous donnons donc la possibilité au développeur de choisir son style d'implémentation. Par le style procédural il peut indiquer que le résultat d'un opérateur n'est qu'une modification partielle d'un ou de plusieurs de ses paramètres. Il s'agit du concept sous-jacent de mutation de données qu'impliquent les paramètres d'entrée-sortie. Nous devons bien sûr lui garantir la validité de l'utilisation de ces paramètres à l'intérieur du code impératif obtenu.

Réutilisation : En effet, en implémentant un opérateur d par une procédure, nous devons permettre l'utilisation de cette procédure par tout code dérivé pour un autre opérateur f, dont l'axiomatisation utilise d.

Soit par exemple :

$$f(x_1, \ldots, x_n) ==> \mathrm{T}_{[x_1,\ldots,x_n]}$$

notre unique équation définissant l'opérateur f. Les variables $x_1, \ldots, x_n$ sont appelées les paramètres formels de l'équation. $\mathrm{T}_{[x_1,\ldots,x_n]}$ est un terme algébrique dont les variables $x_1, \ldots, x_n$ sont liées universellement.

La dérivation vers un langage où la destruction de la valeur des variables est permise demande quelques précautions. Dans une première approche (chapitres 3 à 5), nous ne transformons pas le terme T que nous dérivons (contrairement à la variante du chapitre 6) : nous dérivons chacun de ses sous-termes $t = op(t_1, \ldots, t_n)$ en fonction du résultat de la dérivation de ses propres sous-termes $t_1, \ldots, t_n$. La dérivation est donc valide si et seulement si le texte dérivé pour appliquer op à ses opérandes dans le

domaine concret, a pour paramètres effectifs des représentations correctes des valeurs de $t_1, \ldots, t_n$ (une représentation correcte de la valeur de t est en résultat du texte dérivé, puisque *op* est supposé correctement implémenté).

Or *op* peut être implémenté par une procédure. Soit $t = op(t_1, \ldots, x, \ldots, t_n)$ tel que x est un paramètre formel, et un paramètre effectif en entrée-sortie du code dérivé pour *op*. Après l'exécution du code dérivé de t, la variable x ne représente plus la valeur initiale du paramètre formel, mais la valeur du terme t (ou un élément du produit cartésien valeur de t). Tout terme $t' = op'(t'_1, \ldots, x, \ldots, t'_m)$ sous-terme de T dont le code dérivé est exécuté après celui de t est en conflit d'utilisation de x avec t.

Nous définissons un ordre partiel entre les sous-termes de T suivant l'ordre d'évaluation "obligatoire" (l'ordre de réduction applicatif que nous appelons "de dépendance fonctionnelle", et les ordres entre sous-termes d'opérateurs prédéfinis comme le "*let*"). Puis nous rajoutons un ordre entre sous-termes : un terme dont la dérivation de l'opérateur en racine utilise un paramètre précède un terme dont la dérivation de l'opérateur modifie le même paramètre.

Ces différents ordres peuvent ne pas être compatibles (existence de cycles dans le graphe orienté qui les représente). Par un algorithme de décoration de graphe qui peut être vu comme un raffinement des pré- et post-conditions de chaque sous-terme, nous déterminons les ordres supplémentaires à supprimer pour que tous les ordres restants soient compatibles, et que les duplications de sauvegardes qui sont alors ajoutées soient de coût minimal.

Nous proposons également :

- une modification de cet algorithme de manière à pouvoir attribuer des destinations aux résultats des sous-termes et ainsi éviter des affectations suplémentaires pour les paramètres formels en sortie,
- une optimisation qui consiste cette fois-ci à modifier le terme T avant d'appliquer les algorithmes précédents, pour introduire des mutations à la place de constructions sur des sélections.

Organisation du document

Dans le chapitre 1 nous décrivons le langage de spécification ainsi que celui de programmation qui sont choisis pour être sujets de notre travail. Nous dégageons les concepts différents pour chacun qui font l'intérêt et la difficulté d'une traduction d'un formalisme abstrait de description vers un langage de réalisation concrète.

Dans le chapitre 2 nous montrons les différentes approches existantes de cette transformation, ou la "manipulation" d'objets des deux domaines lorsqu'un lien formel entre la spécification et l'implémentation existe.

Les chapitres suivants proposent une méthode de dérivation de spécification algébrique vers des programmes impératifs :

Dans le chapitre 3 nous définissons le cadre de notre travail. Nous commençons par indiquer ce qu'est intuitivement le lien entre une spécification algébrique et une de ses implémentations impératives. Nous l'appelons lien d'implémentation, mais également lien de dérivation lorsqu'il sert à dériver automatiquement le paquetage (spécification

et corps de programmes impératifs) correspondant à une spécification algébrique. Sont décrits : les problèmes ponctuels à résoudre, le formalisme permettant de définir le lien d'implémentation, et son utilisation pour appliquer dans le domaine concret l'implémentation d'un opérateur à la représentation de ses opérandes. Après l'énumération des outils nécessaires, le principe général de dérivation est défini.

Dans le chapitre 4 nous proposons un algorithme de dérivation de l'axiomatisation d'un opérateur en un code de programme impératif. Nous montrons une résolution des conflits d'utilisation des paramètres formels dans le code dérivé, le coût d'une résolution locale, ainsi qu'un algorithme pour une résolution globale de moindre coût en duplications de données. Cette résolution se fait par la recherche d'un ordre optimal d'exécution des sous-termes de T dans le domaine concret. Lorsque des cycles existent dans le graphe orienté représentant cet ordre, nous calculons l'ensemble des coupures, de sorte que le code engendré, qui ne peut donc être "optimal" (sans duplication de données) soit de coût moindre en duplications. Nous produisons ensuite un texte impératif à partir du graphe orienté sans cycle (d.a.g.).

Dans le chapitre 5 nous proposons une dérivation systématique des constructeurs de la signature d'une sorte. Le lien d'implémentation ainsi que les fonctions ou procédures (duplications, sélections, tests, mutations) nécessaires à la dérivation de tout opérateur dont le domaine ou le rang inclut cette sorte, sont construits automatiquement. Nous reprenons également dans notre formalisme les optimisations connues de codage des types abstraits.

Dans le chapitre 6 nous proposons plusieurs variantes de l'algorithme de dérivation proposé aux chapitres 3 et 4. Il s'agit d'optimisations qui exploitent une vue "flots de données" que les paramètres formels en entrée-sortie permettent de supposer : les applications successives de modifications partielles sur une même donnée. Ces modifications partielles sont également souvent introduites par une sélection de composants d'une structure avant sa reconstruction quasi identique en partie droite d'une règle de réécriture. Nous montrons que des mutations peuvent remplacer des constructions coûteuses, et que les algorithmes peuvent en tenir compte.

En chapitre 7, nous présentons une implémentation simplifié de l'algorithme du chapitre 4.

Chapitre 1

Spécifications formelles et programmes

Dans ce chapitre préliminaire nous présentons le formalisme de spécification que nous avons choisi pour être source de dérivation et nous poursuivons par la description du langage impératif (langage de programmation "traditionnel") utilisé dans les programmes dérivés. Le formalisme de spécification choisi est celui des spécifications algébriques et plus particulièrement celui du langage LPG [BE86] dont nous traitons un sous-ensemble. La plupart des notions standards que nous reprenons ici peuvent être trouvées dans [LZ74, Gut75, GTWW77, EM85, EM90, Wir90], et [Wir94] est un bon panorama des langages de spécifications algébriques. Le langage de programmation choisi est le langage Ada [ALS87]. Nous présentons pour chaque domaine les concepts de base, la syntaxe, et la sémantique qui lui est associée, et les caractéristiques qui expliquent l'adéquation des deux langages et qui ont motivé leur choix.

1.1 Les spécifications algébriques

1.1.1 Concepts syntaxiques de base

La signature est la déclaration des objets définis dans une spécification algébrique. Elle peut être vue comme "l'interface" de la spécification algébrique (le lien que nous définirons d'ailleurs entre une spécification algébrique en LPG et son implémentation impérative en Ada est essentiellement une correspondance entre les objets de la signature de la spécification algébrique et ceux de l'interface Ada de l'implémentation, à laquelle nous ajoutons d'autres informations).

Définition 1.1: (**signature**) Une signature Σ est une paire (S, Ω) où :

- S est un ensemble fini de noms de sortes,
- Ω est une famille $S^* \times S^+$-indicée de noms d'opérateurs où S^* représente une liste de noms de sortes, et S^+ une liste non vide de noms de sortes.

$$\Omega = \{\Omega_{(u,r)} \mid u \in S^*, r \in S^+\}$$

Ω est partitionnée en deux sous-familles :

C une famille $S^* \times S$-indicée de noms d'opérateurs constructeurs (ou *constructeurs*) :

$$C = \{C_{(u,s)} \mid u \in S^*, s \in S\}$$

D une famille $S^* \times S^+$-indicée de noms d'opérateurs définis :

$$D = \{D_{(u,r)} \mid u \in S^*, r \in S^+\}$$

Etant donné un opérateur op de $\Omega_{(u,r)}$, noté $op : u \text{ -> } r$, nous nommons :

- *profil* le couple (u, r),
- *arité* la liste u. Si l'arité est vide, op est une constante,
- *rang* ou *co-arité* la liste r.

■

Nous appelons "*sorte simple*" une sorte qui n'est pas un produit cartésien de sortes, "*sorte produit cartésien*" lorsqu'elle est un produit cartésien de sortes, et "*sorte produit cartésien simple*" une sorte qui est un produit cartésien de sortes simples. Les spécifications que nous implémentons ne comportent que des *sorte simple* ou des *sortes produits cartésiens simples.*

Définition 1.2: (Σ**-terme**) Soient une signature $\Sigma = (S, \Omega)$, et une famille S-indicée X d'ensembles de variables ($X = \{X_s \mid s \in S\}$, X_s disjoints deux à deux) disjointe des constantes de Σ. L'ensemble des termes de sorte $s \in S^+$ noté $T_s(\Sigma, X)$, est le plus petit ensemble contenant :

- les variables de sorte s : $X_s \subseteq T_s(\Sigma, X)$;
- les constantes de sorte s : $\{op \in \Omega \mid op : \text{ -> } s\} \subseteq T_s(\Sigma, X)$;
- le produit cartésien $(t_1, \ldots, t_n) \in T_s(\Sigma, X)$ pour $s \equiv (s_1, \ldots, s_n)$ et $\forall i \in [1 \ldots n] \cdot (t_i \in T_{s_i}(\Sigma, X))$;
- le terme $op(t) \in T_s(\Sigma, X)$ pour l'opérateur $op : (s_1, \ldots, s_n) \text{ -> } s \ \in \Omega$ et $t \in T_{(s_1,\ldots,s_n)}(\Sigma, X)$.

Un terme qui ne contient aucune variable est dit clos ou fermé. L'ensemble de tous les termes de sorte s, pour tout $s \in S^+$, $\bigcup_{s \in S^+} (T_s(\Sigma, X))$ est noté $T_\Sigma(X)$.

■

Définition 1.3: (Σ**-égalité**) Une Σ-égalité est une paire non orientée de Σ-termes de même sorte, notée $t_1 == t_2$.

■

Définition 1.4: (**Σ-équation conditionnelle**) Une Σ-équation conditionnelle est une Σ-égalité, précédée d'une conjonction de Σ-égalités (la condition), et notée $u_1 == v_1 \wedge \ldots \wedge u_n == v_n \Rightarrow t_1 == t_2$. Les variables y sont quantifiées universellement.
■

Définition 1.5: (**Σ-équation simple**) Une Σ-équation simple (ou Σ-équation) est une Σ-équation conditionnelle sans condition.
■

L'axiomatisation d'un opérateur *op* est un ensemble d'axiomes, écrits sous la forme d'équations conditionnelles dont un des deux termes est de la forme $op(t_1, \ldots, t_n)$.

Définition 1.6: (**Spécification**) Une spécification est une paire $SP = (\Sigma, E)$ composée d'une signature et d'un ensemble d'équations.
■

1.1.2 Sémantique

Pour exprimer la sémantique algébrique d'une telle spécification, nous commençons par le concept de *structure algébrique*. La connexion entre la syntaxe et la sémantique est donnée par une *interprétation*. Il s'agit d'une application $\imath$ de la signature Σ vers une structure $\mathcal{A}$ qui associe à chaque sorte s un ensemble support $\mathcal{A}_s = \imath(s)$ et à chaque nom d'opérateur *op* une fonction $f_{op} = \imath(op)$. Lorsqu'une telle interprétation existe, nous appelons la structure $\mathcal{A}$ une Σ-algèbre. L'application $\imath$ est un homorphisme. Soit $op : (s_1, \ldots, s_n) \text{ -> } (s_{n+1}, \ldots, s_{n+m})$ avec $\imath(op) = f$, alors f est une fonction dont le domaine est $(\imath(s_1), \ldots, \imath(s_n))$ et dont le co-domaine est $(\imath(s_{n+1}), \ldots, \imath(s_{n+m}))$. Bien sûr, l'interprétation des noms d'opérateurs doit être conforme à leur fonctionnalité.

Nous utiliserons *domaine* et *co-domaine* respectivement pour les ensembles des paramètres et du résultat des fonctions.

Définition 1.7: (**modèles**) Une structure algébrique $\mathcal{M}$ est appelée *modèle* d'une spécification $SP = (\Sigma, E)$ s'il existe une interprétation $\imath$ de la signature Σ de SP vers la structure algébrique $\mathcal{M}$ telle que toutes les équations de E soient valides dans $\mathcal{M}$.
■

La classe de tous les modèles de la spécification SP est notée $Mod(SP)$.

Cette notion de modèle suit une tradition mathématique plus ancienne que celle de l'informatique: les mathématiciens parlent "des" entiers sans préciser s'ils sont vus comme un groupe, un anneau, ou un ordre partiel. Différentes sémantiques peuvent donc être adoptées pour une spécification algébrique: chaque valeur de $\mathcal{A}_s = \imath(s)$ est l'interprétation d'un Σ-terme ($\mathcal{A}_s$ est finiment engendrée) et les sémantiques diffèrent selon que $E \vdash t_1 == t_2 - \imath(t_1) = \imath(t_2)$ ou uniquement $E \vdash t_1 == t_2 \Rightarrow \imath(t_1) = \imath(t_2)$.

Le premier cas est celui où la sémantique de $SP = (\Sigma, E)$ est définie par l'algèbre des termes T_Σ quotienté par les équations de E. Ce modèle est nommé *modèle initial* (à un isomorphisme près), et la sémantique qui à une spécification associe son modèle

initial, *sémantique initiale*. Le deuxième cas, où la sémantique est donnée par une classe de modèle est dite *lâche*.

1.1.3 Notre langage de spécification algébrique

Le langage de spécification algébrique que nous utilisons comme source des dérivations est une version *simplifiée* et *légèrement modifiée* de LPG [BE86]. Nous l'appellerons LPG par commodité : les différences étant faibles, il est possible de traduire simplement une spécification de notre version de LPG vers la version standard.

Les unités LPG que nous dérivons sont, soit des *types*, soit des *enrichissements*. Les types sont composés de :

- déclaration de sortes,
- déclaration de constructeurs (pour les sortes précédentes),
- déclaration d'opérateurs définis,
- déclaration de variables avec leurs sortes,
- déclaration d'exceptions,
- d'équations orientées qui définissent les opérateurs précédents,
- de traite-exceptions.

Les enrichissements ne comportent pas de déclaration de sortes ni de constructeurs.

La généricité est un point fort du langage LPG. Les types et les enrichissements peuvent être paramétrés par des sortes et des opérateurs qui respectent une propriété particulière.

Une unité générique de tri de séquences est paramétrée par exemple par l'élément constituant les séquences, qui doit respecter la propriété d'ordre total :

```
Enrichment Trier requires Total_Order[elem,<=,=]
operators
    insert_trie : (elem,seq[elem]) -> seq[elem]
    trier : seq[elem] -> seq[elem]
variables
    x, y : elem
    l    : seq[elem]
equations:
1: trier(nil) ==> nil
2: trier(x<+l) ==> insert_trie(x,trier(l))
3: insert_trie(x,nil) ==> x<+nil
4: insert_trie(x,y<+l) ==>
            if x<=y then x<+(y<+l)
            else y<+(insert_trie(x,l))
            endif
end Trier
```

Nous nous limitons à un seul paramètre générique sans propriété particulière: il satisfait la propriété `Formal_Sort`. L'unité LPG est aisément implémentée par un paquetage générique paramétré par un type privé. Le lien sorte formelle paramètre avec type privé paramètre est alors ajouté au contexte de dérivation.

ätendre cette généricité aux opérateurs impose de fixer la spécification (style et profil) des procédures ou fonctions paramètres, à partir de la sorte et de l'arité des opérateurs. Ces informations (sortes et arités) sont contenues dans la définition de la "propriété" requise par l'unité LPG, et la spécification impérative correspondante doit être donnée par l'utilisateur. Cette extension ne sera pas détaillée dans ce mémoire, mais deux solutions sont possibles:

- ne définir le lien d'implémentation d'un module LPG générique et ne faire sa dérivation qu'au moment où il est instancié;
- effectuer les dérivations d'un module LPG générique selon toutes les implémentations possibles (toutes les interfaces possibles) de l'opérateur paramètre.

Les opérateurs définis doivent être complètement spécifiés dans une même unité. Ils ont pour domaine un produit cartésien de sortes simples (éventuellement vide), et pour co-domaine, une sorte simple, ou un produit cartésien de sortes simples.

Soit $t = op(t_s, \ldots, t_n)$ un Σ-terme, nous appelons *op* l'*opérateur principal* de t. Les équations orientées définissent l'axiomatisation des opérateurs définis: soit $t \Longrightarrow t'$ une Σ-équation, t a pour opérateur principal un opérateur défini, dont les opérandes ne sont construits qu'avec des constructeurs et des variables. Nous n'avons pas d'équations sur les constructeurs, c'est-à-dire que nous n'avons pas d'équation $t \Longrightarrow t'$ avec un constructeur en opérateur principal de t.

Les Σ-équations penvent être transformées en des Σ-équations conditionnelles car l'opérateur "*if-then-else*" est un opérateur prédéfini dont le test booléen b peut devenir une condition $b == true$ d'une équation conditionnelle. L'inverse ne peut être fait que si l'on possède une axiomatisation de l'égalité pour les sortes de la partie condition des équations conditionnelles.

Les opérateurs prédéfinis :
Soient les termes LPG $t_1,\ldots,t_n$ des termes dont la sorte est une sorte simple, et $t'_1,\ldots,t'_n$ dont la sorte est, soit une sorte simple, soit un produit cartésien de sortes simples. Ils sont construits à partir de constructeurs, d'opérateurs définis, ou d'opérateurs prédéfinis.

Les opérateurs prédéfinis sont:

- le produit cartésien "`(`t_1`,...,`t_n`)`",
- l'égalité "t_1 `=` t_2",
- la conditionnelle "`if` t_1 `then` t'_1 `else` t'_2 `endif`"
  ou "`if` t_1 `then` t'_1 `endif`" avec t_1 de sorte *bool*,
- la définition de variable(s) locale(s): "`let` v_1 `=` t_1 `in` t'_1 `endlet`" ou "`let (`v_1`,...,`v_n`) =` t'_2 `in` t'_3 `endlet`" avec $v_1,\ldots,v_n$ des variables dont la sorte

est une sorte simple, et t'_2 un terme dont la sorte est un produit cartésien de sortes simples.

Les exceptions dont seuls les noms sont déclarés dans la section `Exceptions` sont définies par leurs *déclencheurs* et *traite-exception* dans une section `Conditions`. Elles permettent de décrire le comportement d'un opérateur qui spécifie une fonction partielle lorsqu'il est appliqué hors de son domaine d'application. Elles sont définies par un triplet (*op(paramètres)*, *test*, *idf_exc*) où *op* est un opérateur déclaré dans l'unité courante, les *paramètres* sont un motif ("pattern"), *test* une expression booléenne optionnelle et *idf_exc* un nom d'exception :

op(*paramètres*)
`if` *test* (optionnel)
`raises` *idf_exc*(*paramètres2*)

Lorsque les paramètres effectifs de *op* coïncident ("match") avec *paramètres*, et que l'expression *test* a pour valeur *vrai*, alors l'exception de nom *idf_exc* est déclenchée.

Le mécanisme de propagation est le même que celui de Ada dans le cas séquentiel. Les exceptions récupérées pendant l'évaluation d'un terme *op(paramètres)* sont définies également dans la section `Conditions` de la manière suivante :

op(*paramètres*)
`when` *idf_exc*(*paramètres2*)
`raises` *idf_exc*(*paramètres3*)

ou

op(*paramètres*)
`when` *idf_exc*(*paramètres2*)
`==>`*expression*

Les expressions *paramètres* doivent coïncider avec n'importe quelles valeurs et sont donc des variables. Dans le premier cas une exception est "re-levée", et dans le second cas, *op(paramètres)* prend pour résultat *expression*.

1.2 Les programmes impératifs

Nous désignons par langage impératif un langage de programmation "traditionnel", avec déclaration des variables (dont l'ensemble définit l'*état* d'un programme), et dont les programmes sont des suites d'instructions qui modifient l'état. L'instruction caractéristique est l'affectation destructrice d'une valeur à une variable.

Dans un langage purement fonctionnel, nous ne manipulons que des valeurs. Un programme désigne une fonction qui calcule une valeur à partir de ses opérandes. Chaque pas d'exécution d'un programme fonctionnel, consiste en une application d'une fonction sur les valeurs de ses opérandes déjà calculées. L'ordre dans lequel sont calculés tous les sous-termes du terme qui constitue le programme fonctionnel, est appelé "*ordre de réduction*".

Un programme impératif peut comporter à la fois des appels de fonctions et des appels de procédure. Dans un langage impératif, une fonction peut évidemment ne

faire aucune affectation destructive d'une variable, ni aucun appel à une procédure. Ce *style* de programmation est alors appelé *fonctionnel*, et par opposition, le style de programmation d'un langage impératif qui n'est pas fonctionnel est appelé procédural.

Chaque pas d'exécution d'un programme impératif, de style procédural, consiste en une application d'une instruction qui modifie les valeurs de l'ensemble des variables.

Nous avons choisi le langage Ada comme représentant des langages impératifs car il possède de nombreux concepts proches de ceux du langage de spécification LPG :

- la modularité : l'unité de compilation est le sous programme (fonction ou procédure) ou le paquetage. Les types Ada qui ne sont pas prédéfinis et les paquetages référencés dans une unité doivent être importés avant d'être utilisés. En LPG, les types algébriques ou enrichissements connus sont ceux qui sont prédéfinis, ou compilés ou chargés préalablement dans l'évaluateur ou le résolveur ;

- la généricité : une unité Ada peut avoir une liste de paramètres génériques. Un paramètre générique est soit un objet formel (une constante), soit un type, soit une fonction ou une procédure. Les paramètres génériques du langage LPG sont une liste de sortes et d'opérateurs préfixés par la propriété qu'ils respectent (les propriétés sont définies séparément par un ensemble d'axiomes) ;

- les exceptions : elles sont déclarées en début de l'unité Ada, levés par l'instruction `raise` dans les corps de programmes, ou par une violation des limites des domaines des arguments. Elles peuvent être récupérées dans un traite-exception fourni par le programmeur en fin d'instruction bloc ou de corps de programme ou de paquetage. Les conditions pour lesquelles les exceptions en LPG sont levées, et récupérées, sont définies dans une rubrique indépendante du texte de la spécification. Elles sont définies en fonction des valeurs des opérandes, ce qui correspond à des exceptions Ada levées en début de corps de programme, et récupérées en fin. Le mécanisme de propagation est le même que celui du langage Ada dans le cas séquentiel.

Nous n'allons pas faire une description détaillée du langage Ada, mais énumérer les concepts syntaxiques de base que nous utilisons. Une description complète du langage Ada se trouve dans le manuel de référence [ALS87].

L'unité de compilation en Ada est soit un paquetage (une collection de types, de données, de fonctions et de procédures), soit une fonction ou une procédure. Dans nos dérivations, nous produisons un paquetage qui implémente soit un type algébrique, soit un enrichissement.

Ce paquetage peut être générique, mais nous nous limitons à un paramètre générique égal à un type Ada. Un paquetage est composé de deux entités rangées dans deux fichiers :

- La spécification,

- Le corps.

Dans la spécification du paquetage, nous avons la déclaration et définition des types, la déclaration des fonctions et procédures. Dans le corps nous avons les fonctions et procédures avec leur corps de programme.

Une fonction déclarée par :

`function foo (`v_1 : typ_1; ... ; v_n : typ_n`) return` typ`;`

désigne les noms $v_1, \ldots, v_n$ de ses paramètres formels, de types respectifs $typ_1, \ldots, typ_n$, et retourne une valeur de type typ. Une procédure déclarée par :

`procedure bar(`v_1 : mod_1 typ_1; ... ; v_n : mod_n typ_n`);`

désigne comme informations supplémentaires les modes $mod_1, \ldots, mod_n$ de ses paramètres formels, qui ont pour valeur `in`, `out` ou `in out`. Les modes des paramètres formels d'une fonction sont implicitement `in`.

Le corps de programme des fonctions ou procédures Ada est constitué d'une séquence d'instructions parmi :

- Les instructions prédéfinies, dont principalement :

  l'affectation : "*variable*`:=`*expression*`;`",

  le test : "
  ```
  if expression-booléenne then
       instruction ; instruction ; ...
  else
       instruction ; instruction ; ...
  end if; ",
  ```
  ou
  ```
  " if expression-booléenne then
       instruction ; instruction ; ...
    end if; "
  ```
  le bloc : "`begin` *séquence-d'instructions* `end`",

- L'appel de procédure $bar(e_1, \ldots, e_n)$`;` où les expressions $e_1, \ldots, e_n$ sont appelées paramètres effectifs.

Les appels de fonction font partie des expressions, parmi lesquelles nous trouvons les instructions standard de manipulation des données (dont nous utilisons essentiellement l'accès aux composants indicés, composants sélectés, et le déréférencement).

Certaines expressions peuvent recevoir une valeur en étant en position d'un paramètre de sortie ou entrée-sortie (`out` ou `in out`), ou en partie gauche d'une affectation. Elles conservent cette valeur tant qu'une autre valeur ne leur est pas affectée (directement ou indirectement par l'utilisation d'un pointeur sur un composant de cette expression, mais le code engendré par nos transformations n'utilise pas cette deuxième possibilité). Une telle expression a le droit d'être la *destination* d'un calcul, et est nommées "*expression-gauche*". En Ada, ce sont :

- les variables globales d'un paquetage,
- les variables locales d'une fonction ou procédure,
- les paramètres formels en sortie ou entrée-sortie d'une procédure,
- pour une expression-gauche `v`, ce sont les composants :
  - indicés `v[indice]`,

- sélectés `v.champ`,
- référencés `v.all`.

Le prédicat *LeftExpr* est vrai pour toutes les expressions que nous acceptons comme expressions-gauches dans notre méthode de transformation. Le chapitre 3 montre son utilisation dans un pas de transformation, et dans le chapitre 4 nous décrivons une méthode de transformation pour laquelle seules les variables locales d'une fonction ou procédure, et les paramètres formels en sortie ou entrée-sortie d'une procédure sont acceptées comme expressions-gauches.

Chapitre 2

Des spécifications formelles aux programmes

Dans ce chapitre, nous présentons différentes approches qui permettent d'obtenir à partir d'une spécification formelle, soit directement un code exécutable dans un langage de programmation, soit une autre spécification, dans un domaine plus concret, qui permet de décrire la réalisation de la spécification.

Nous décrivons essentiellement deux approches, que nous pouvons différencier par la quantité de "valeur ajoutée" qu'il faut apporter en plus de la spécification formelle :

Développement : la source du développement est la spécification formelle (éventuellement algébrique) qui ne décrit que l'essentiel du calcul ou des valeurs. Elle peut d'ailleurs décrire une classe de modèles et non un seul. Le terme de spécification conserve son sens : "spécifier, c'est décrire un problème sans décider prématurément de la façon dont il sera résolu". Le processus de développement n'est pas automatisable car il fait partie du processus de spécification,

Dérivation : il y a suffisamment d'informations dans la spécification pour pouvoir en déduire un programme exécutable dans un domaine concret (éventuellement du niveau d'un langage de programmation). Cette approche peut avoir pour adage "spécifier, c'est (presque) programmer".

Les dérivations se font à sémantique constante, alors que les développements permettent l'ajout d'information. Néanmoins, les frontières de cette classification[1] ne sont pas nettes puisque les dérivations exigent pour la plupart une intervention humaine (choix de domaine de représentation, choix de profil, "eureka", ...). L'accent est mis, d'un côté sur l'automatisation, et de l'autre sur l'interaction.

Les articles fondateurs du domaine sont issus des travaux de J.V. Guttag [Gut75, Gut77], N. Wirth [Wir71] et R.M. Burstall et J. Darlington [Dar75, BD77, Dar83].

Une synthèse et classification des différentes motivations, approches, techniques et systèmes de dérivation peut être trouvée dans [PS81] par H. Partsch and R. Steinbruggen.

1. classification que H. Partsch et R. Steinbruggen nomment "classification selon les intentions".

2.1 Développements

Dans [BLY93] Valdis Berzins, Luqi et Amiram Yehudai montrent l'importance des développements formalisés dans le cycle de vie d'un gros projet. Dans le cas qui nous intéresse, les transformations prennent place dans le cycle de vie au même niveau que la décomposition hiérarchique en modules, par la transformation d'une spécification en un module exécutable. Le cadre proposé permet de conserver un historique du développement, d'en avoir différentes alternatives, de développer en parallèle, d'améliorer la réutilisabilité et la réutilisation, d'être un outil tutoriel.

2.1.1 L'implémentation abstraite

Il s'agit de ce que nous pouvons appeler des "transformations verticales" dans un même formalisme. En effet, les domaines source et destination du développement sont au même niveau de langage. Le codage des types abstraits de données a été beaucoup étudié. Les premières références sur le sujet sont [Hoa72], qui s'appuie sur la notion d'abstraction, et [GTWW75, GTW78] du groupe ADJ (composé de J. Goguen, J. Thatcher, E. Wagner et J. Wright), basées sur la notion de représentation.

D'autres travaux ont porté sur la définition de la sémantique algébriques des implémentations, par la définition d'opérateurs de transformation [EKP80, EKMP82, ST87, Pep91]. Le principe en est le suivant :

Le point de départ est une spécification formelle A donnée par une présentation (Σ, E), la représentation usuelle de la classe initiale des Σ–algèbres qui satisfont E est l'algèbre des termes $T_\Sigma / \equiv_E$ notée $T_{\Sigma,E}$.

La méthodologie consiste, de manière simple à :

- Construire ΔB en partant d'un type abstrait B existant : sortes et opérateurs qui implémentent ceux de Σ ($\Sigma = sig(A)$).
- Identifier les sortes et opérateurs construits à ceux de Σ en appliquant la dérivation $|_\sigma$ où σ est un morphisme de Σ vers $sig(B+\Delta B)$. Nous obtenons $(B+\Delta B)\ |_\sigma$.
- Restreindre les domaines des algèbres obtenues à ceux que l'on peut atteindre par les opérateurs de Σ. Nous obtenons $R((B + \Delta B)\ |_\sigma)$.
- Identifier les termes de la représentation qui représentent le même terme du type abstrait de A. Nous obtenons $(R((B + \Delta B)\ |_\sigma))/_\equiv$ notée C.

Il faut ensuite prouver que l'implémentation réalisée représente bien le type abstrait d'origine. Nous retiendrons les critères de réalisabilité (tout élément du domaine abstrait doit être représenté), de consistance (deux éléments distincts du domaine abstrait ne doivent pas être représentés par le même élément du domaine concret) et d'homomorphisme (les opérateurs du domaine concret qui implémentent des opérateurs du domaine abstrait doivent leur être homomorphes).

L'inconvénient de cette approche est que la correction des implémentations est difficile à réaliser en pratique.

Donald Sannella et Andrzej Tarlecki, dans [ST87], généralisent la notion d'implémentation abstraite de [EKMP82]. Ils considèrent que les spécifications définissent une

signature et une classe de modèles sur cette signature. Ils définissent des opérateurs "constructeurs" qui construisent de nouvelles spécifications à partir des spécifications arguments. C'est ainsi que dans le cas ci-dessus, on a une implémentation de A par la spécification: "quotient (restrict (enrich B by ΔB) by σ) by $\equiv$" où les identificateurs sont des opérateurs sur des spécifications. Ils introduisent également la notion d'équivalence "comportementale" entre des algèbres qui est une autre façon de définir une sémantique observationnelle.

Didier Bert et Sadik Sebbar utilisent dans [BS91] le méta-langage DEVA pour décrire comment réaliser des développements basés sur la théorie algébrique, comme la représentation de type de données ou la synthèse de définitions de fonctions dans un domaine de représentation. DEVA est un langage qui allie le lambda-calcul typé, la déduction naturelle et la logique constructive. Il est basé sur un système de types dépendants où les objets et les types sont dans une même classe. Il fournit un seul cadre pour exprimer les spécifications, les programmes, ou les développements, ou d'une manière "isomorphe", les théorèmes d'une théorie, les preuves et les tactiques. Dans la notation DEVA, un développement est un objet sur lequel il est possible de réaliser des opérations (abstraction, application, composition, etc.).

Une approche logique a été décrite dans [MVS85].

Le cadre de l'implémentation abstraite a été unifié de manière élégante par l'observabilité [Ber86, BB92, Kna93]. Cette approche permet de définir des critères de correction de l'implémentation dans les différentes sémantiques algébriques (initiale, lâche, lâche hiérarchique).

Une bonne synthèse des différentes approches et des différents travaux existants dans le domaine peut être trouvée dans [Ore93]. F. Orejas effectue également dans cet article, une clarification de la notion de transitivité de l'implémentation. Il montre que l'ordre d'application des opérateurs qui construisent une implémentation peut conduire à la création, au niveau de la sémantique algébrique, de valeurs qui sont en-dehors du domaine de représentation ("*junk*") : l'identification suivie de la restriction, contrairement à l'ordre inverse, peut avoir ces problèmes de "junk". Par contre, cet ordre de composition facilite la preuve de la correction de l'implémentation (alors équivalente à la consistance hiérarchique), ainsi que, au niveau programmation, de l'écriture du prédicat d'égalité.

Le point de vue de F. Orejas est que le manque de transitivité du concept d'implémentation n'est pas réellement important, tant que l'on a affaire aux spécifications (et non aux programmes), et tant que le langage de programmation satisfait les propriétés adéquates.

2.1.2 Le raffinement

Il s'applique surtout dans le cas des "spécifications orientées modèles" : la spécification d'une classe d'outils ou de problèmes est précisée au fur et à mesure du raffinement. Le langage B [Abr91] par exemple, possède des caractéristiques explicites de raffinement qui conduisent à changer la représentation de l'état des machines abstraites et à réduire l'indéterminisme. Un autre exemple en est le langage VDM [Jon86].

Un des travaux fondateurs est celui de Niklaus Wirth [Wir71] qui illustre par un

exemple (la résolution du bien connu "problème des 8 reines") comment effectuer un raffinement, à la fois des données, et des tâches spécifiées. La représentation des données est raffinée de manière à faciliter l'application des tâches. Les tâches à appliquer aux données sont raffinées parallèlement, pour suivre le raffinement des données, et pour être décomposées en sous-tâches, jusqu'à être exécutables dans un langage de programmation.

Edsger W. Dijkstra présente dans [Dij76] un travail mettant en avant la rationalisation des développements. Les spécifications de programmes sont données par des pré-conditions et post-conditions que doivent vérifier ces programmes. Cela signifie que le prédicat pré-condition est la condition minimale pour que le programme s'exécute, se termine, et que le prédicat post-condition soit vérifié. Edsger W. Dijkstra propose des heuristiques pour construire le programme à partir du prédicat post-condition, qui permet à partir de là d'établir la pré-condition comme "plus faible pré-condition" (*weakest preconditions*).

Un ensemble de travaux de même approche (basés sur un sémantique à base de plus faibles pré-conditions) peut être trouvé dans [MGRV92]. Le langage utilisé à la fois pour le domaine abstrait et le domaine concret est une extension du langage à commande gardée de Dijkstra. On y montre une nouvelle technique de raffinement, qui simplifie les dérivations en permettant que des étapes intermédiaires violent la loi du "miracle exclu" de Dijkstra [Dij76] («tout programme qui se termine possède un état», ou «il n'existe pas de pré-condition minimale pour qu'un programme se termine et qu'alors aucun prédicat ne soit vérifié» : pour un programme P, $wp(P, false) = false$).

David Gries présente dans [Gri92] une notion du raffinement des données d'un programme procédural. Les raffinements sont définis par un remplacement de variables d'un programme par d'autres variables, avec le remplacement adéquat des expressions et instructions du programme pour obtenir un programme équivalent. La relation entre les variables remplacées et les variables de remplacement, qui contiennent une représentation (un codage) des valeurs des premières, est appelée "invariant de couplage". Chaque remplacement d'une instruction appliquée à une variable x par une instruction appliquée à une variable y qui contient une représentation de la valeur de x, doit conserver cet invariant. La sémantique des programmes est donnée par des plus faibles pré-conditions et les raffinements génèrent des obligations de preuves pour assurer leur validité.

2.1.3 Larch : liens spécifications – programmes par preuves

Le projet Larch [GH93] peut être vu comme un effort pour remplir le fossé entre les langages de spécification algébrique, et les langages de programmation impérative. Il adopte une approche à deux étages : les abstractions sont spécifiées à l'aide du "Larch Shared Language" (LSL), qui est principalement un langage de spécification algébrique, tandis que les transformations d'états sont spécifiées à l'aide du "Larch Interface Language" (LIL), qui suit le paradigme des pré- et post-conditions.

Le Larch Shared Language est un moyen de décrire des types abstraits et des théories qui sont adaptés à la spécification des programmes. Ses composants appelés traits, sont utilisés pour écrire les pré- et post-conditions du Larch Interface Language,

non pour être implémentés. Par contre, ce langage commun permet d'utiliser les théories du LSL pour faire la preuve que les programmes respectent les pré- et post-conditions qui les spécifient.

2.1.4 Techniques par "plans de travail"

Ce sont des techniques où la spécification d'origine est incomplète, et où les applications successives d'opérateurs de transformation génèrent à chaque étape une nouvelle spécification, plus précise, jusqu'à obtenir un degré de complétude suffisant (programme exécutable, par exemple).

Les motivations du projet PROPLANE (anciennement PLANO) [Sou93, SD95] sont plus dans la connaissance de l'historique du développement que dans le résultat lui-même. En effet, la valeur ajoutée lors d'un développement vient plus de l'utilisateur que de l'outil, et la conservation des choix d'application des opérateurs de développement apporte :

- un support à la documentation du développement, qui facilite la compréhension et l'évolution de la spécification;
- un support à la réutilisation. Le développement peut être rejoué grâce à la mémorisation des décisions prises lors des différentes étapes. Les justifications des choix sont une aide non négligeable pour :
  - déterminer l'adéquation à appliquer tout un développement à une nouvelle spécification d'origine;
  - réutiliser le produit lors de l'évolution d'un logiciel, en ne faisant évoluer que les étapes où les choix faits ne sont plus valides;
- une indépendance par rapport au langage support du résultat.

Le but avant tout organisationnel : il s'agit d'intégrer des outils de développement et de coordonner les activités des développeurs en fonction de l'état courant du développement. Mais malgré l'affirmation précédente que la valeur ajoutée vient plus de l'utilisateur que de l'outil (même si l'utilisateur ne peut apporter cette valeur ajoutée sans cet outil), des stratégies et tactiques sont définies pour aider l'utilisateur. En effet, le nombre d'opérateurs applicables à chaque étape est élevé.

Ces informations textuelles sont composées de : une situation (conditions d'application), une heuristique (qui précise les critères d'application, et l'orientation choisie), une motivation (justification de l'application) et un guide de développement (suggestion d'opérateurs pour la poursuite du développement).

Un exemple d'opérateurs qui permettent d'appliquer un processus de développement générique (définir un type) et l'instanciation de ces opérateurs sur une spécification (un réseau téléphonique) sont décrits dans [DS93].

F. Losavio propose dans [Los91] une méthode de construction de programmes Ada à partir de leurs interfaces. Elle est basée sur la décomposition de types abstraits algébriques.

Une bibliothèque de spécifications contient un ensemble de types abstrait algébriques d'usage courant, écrits en PLUSS [Bid89, Gau84]. Pour chacun d'eux, un *schéma de décomposition abstrait* énumère l'ensemble des sélecteurs applicables à un objet de la sorte du type abstrait.

Une bibliothèque de programmes contient un ensemble de programmes (Ada) qui implémentent les opérations des types abstraits algébriques.

Une bibliothèque de *schémas de décomposition concrets* contient pour chaque type algébrique les fonctions Ada qui implémentent les sélecteurs des schémas de décomposition concret.

Une bibliothèque contient des *stratégies de construction*, c'est-à-dire des schémas de programmes qui utilisent les variables résultant de l'application d'un schéma de décomposition concret, pour un calcul caractéristique (combinaisons, tests, etc).

La méthode de développement d'un programme à partir d'une interface Ada appelée "problème d'intérêt" est l'application d'un schéma de décomposition concret sur un paramètre choisi de ce problème d'intérêt, et d'appliquer sur le résultat ainsi que les autres paramètres, une stratégie de construction de programmes. Nous pouvons compléter manuellement le squelette de programme obtenu, ou bien appliquer à nouveau la méthode décrite sur une des fonctions anonymes que la stratégie de construction a utilisée.

2.1.5 Synthèse

Nous distinguons les travaux qui se fondent sur la logique classique [MW71, MW81, Pot88] de ceux qui sont basés sur le typage et la logique constructive.

La logique de Hoare a été développée pour pouvoir démontrer quelle est la signification des programmes. La sémantique est décrite par des prédicats sur l'état (les variables) des programmes. Des règles énoncent comment les différentes constructions de langage transforment ces assertions (calcul des plus faibles pré-conditions).

D'un autre côté, les objets et les programmes peuvent être décrits par un typage suffisamment puissant pour en décrire le sens.

Ces deux sémantiques permettent de considérer un programme comme une preuve d'une formule logique (un théorème) et donc d'assimiler la construction d'un programme à la recherche d'une preuve. Ceci est démontré par l'isomorphisme de Curry-Howard [How69], qui montre la correspondance entre les systèmes de preuve en déduction naturelle et les λ-calcul typés. L'extraction d'un programme à partir d'une preuve d'un théorème est possible si celle-ci est constructive, c'est-à-dire si nous sommes capables de calculer tous les objets qu'elle met en œuvre. L'axiome du tiers exclu : $A \vee \neg A$ est une proposition qui ne donne aucune information sur la prouvabilité de A ou la prouvabilité de $\neg A$. Cet axiome de la logique classique est exclu de la logique intuitionniste, où il est possible, à partir d'une preuve de $A \vee B$, de construire, soit une preuve de A, soit une preuve de B. Ce caractère calculatoire des preuves intuitionnistes est mis en évidence par la réalisabilité, notion introduite par S. Kleene en 1945 [Kle52]. Il existe différents formalismes constructivistes [TvD88], [Bee85], [Hey66], mais l'ouvrage de référence dans le domaine de la réalisabilité est celui de A.S. Toelstra [Toe73]. La réalisabilité est le cadre théorique pour étudier l'extraction de programmes corrects à partir de preuves constructives. Elle apparaît de manière essentielle dans des systèmes

tels que Nuprl de R. Constable [CABC86], "Automath" de Martin-Löf [ML80], le calcul des constructions de Th. Coquand et G. Huet [CH87], [PM89] et AF-2 de J.-L. Krivine [KP87].

2.2 Dérivations

L'évolution des logiciels est rendue difficile pour les projets de taille réelle par la recherche de l'efficacité dans le codage. Le passage d'un programme simple mais peu efficace à un programme optimisé est une transformation à sémantique constante, qui introduit des contraintes supplémentaires. Il est donc avantageux de pouvoir effectuer les évolutions d'un système à un haut niveau d'abstraction, sans se préoccuper de l'efficacité, qui est obtenue par des dérivations appliquées ensuite.

Vandervoorde, dans [Van94] montre qu'à partir des spécifications nous pouvons définir de nouvelles optimisations, qui rendent l'interface générale plus efficace à utiliser, ainsi qu'avoir une structure d'accueil adéquate pour les optimisation classiques.

2.2.1 Transformations

Nous utiliserons le terme de "texte" pour désigner un programme ou une spécification. La catégorie des dérivations que nous appelons *transformations* est l'ensemble des opérations qui opèrent une modification d'une partie d'un texte pour obtenir un texte équivalent, sans apport d'informations supplémentaires.

Parmi ces opérations se trouvent les techniques de "*pliages* et *dépliages*" très utilisées dans les transformations de programmes à des fins d'optimisation : récursification ou dérécursification, combinaisons de boucles, etc. Ces opérations sont décrites dans [BD77, Dar83] ou dans [Sek88, SF86, PP88, AFQ86] pour les programmes logiques (manipulations de clauses de Horn).

Robert Gabriel présente dans [Gab91] comment des techniques de transformation de programmes peuvent être définies en DEVA. Un exemple du schéma de "mouvement de parenthèses" pour l'optimisation des fonctions récursives linéaires, est montré. La preuve de la correction du schéma et les théories nécessaires sont données en DEVA.

2.2.2 Dérivations du Fonctionnel vers l'Impératif

Les nombreux avantages de chaque domaine : programmation fonctionnelle et programmation impérative ont motivé de nombreuses recherches sur la définition de systèmes ou langages qui allient les deux styles. David K. Gifford et John M. Lucassen dans [GL86] montrent les différents avantages et inconvénients des deux styles, qu'ils associent en un langage qu'ils qualifient de *courant* (fluent) où des invariants sur des valeurs sont comparés à des invariants sur des effets-de-bord.

Les spécifications algébriques ont la caractéristique d'être fonctionnelles. Les langages fonctionnels sont reconnus pour leur puissance d'expression et leur sémantique relativement directe. Ils permettent cependant des implémentations différentes, incluant l'ordre de réduction applicatif (AOR, appel par valeur dans les langages de programmation conventionnels) ou normal (NOR, appel par nécessité; le résultat d'un calcul effectué en utilisant l'appel par valeur diffère de celui obtenu avec l'appel par nécessité,

seulement si l'expression en argument produit des effets de bord), qui correspondent à l'évaluation paresseuse ou stricte.

La transformation d'une spécification fonctionnelle en un programme impératif, et l'utilisation de procédures existantes introduisent la notion de passage de paramètres : Caroll Morgan [Mor90] étudie la notion de raffinement de programmes qu'il note $S \sqsubseteq T$ pour "*S est raffiné par T*". Il utilise une sémantique basée sur les plus faibles préconditions (*wp*), avec pour règle d'application de procédure, la règle de copie d'Algol (insertion du corps de la procédure avec substitution des paramètres renommés). Il donne les règles de substitution capables de décrire le passage de paramètres par valeur, résultat, et valeur-résultat (in, out, in-out). Mais ces règles sont limitées au cas de la substitution simple par nom et ne traitent pas le cas général de la substitution par nom, ni de celle par variable. Ce cadre permet déjà de montrer la perte de monotonie lors de l'utilisation d'une procédure (ne plus avoir $P1 \sqsubseteq P2 \Rightarrow P1[x\backslash T] \sqsubseteq P2[x\backslash T]$) et le phénomène d'alias qui apparaît (lorsque deux noms distincts désignent la même variable).

Le cas de la substitution par variable est décrit par D. Gries dans [Gri82] comme une forme efficace de l'appel avec passage par valeur-resultat, sans affectations initiales ni finales des paramètres effectifs. Cette extension n'est valable que si les paramètres effectifs occupent des emplacements en mémoire disjoints (sans partage).

Approche de Huimin Lin

Le contenu principal de l'article [Lin93] de Huimin Lin est de proposer une nouvelle technique de vérification formelle de la correction de l'implémentation de spécifications algébriques dans un langage de la programmation impérative traditionnelle. Le domaine dans lequel Huimin Lin exprime les propriétés (monotonie, régularité) de ses implémentations est celui de la logique de Hoare, et se base sur les définitions et utilisations des "plus faibles préconditions". Les travaux dont il s'est inspiré sont ceux de Caroll Morgan [Mor90], E.W. Dijkstra [Dij76], D. Gries [Gri82] et C.B. Jones [Jon86].

Soit P un programme et R un prédicat, $wp(P, R)$ est le prédicat le plus faible tel que s'il est vérifié sur le domaine de P alors R est vrai après l'exécution de P. C'est-à-dire : soit $B = wp(P, R)$, alors $\{B\}P\{R\}$ et quel que soit le prédicat C tel que $\{C\}P\{R\}$ alors $C \Rightarrow B$.

Huimin Lin présente :

- une méthode d'implémentation d'une spécification algébrique par un programme impératif, et une technique de dérivation d'un terme dans cette implémentation,
- un critère de correction de la relation d'implémentation,
- une paramétrisation du critère de correction d'implémentation, dans une approche observationnelle, ce qui permet de tenir compte des différentes sémantiques que l'on peut avoir pour les spécifications algébriques (sémantique initiale ou lâche, hiérarchique, ...),
- une définition du raffinement des implémentations, ce qui permet de montrer que le raffinement d'une implémentation est aussi une implémentation : le programme

P a pour raffinement Q, nous notons $P \sqsubseteq Q$ si et seulement si quelle que soit la post-condition R, $wp(P,R) \Rightarrow wp(Q,R)$.

- un lien entre les spécifications à base de pré- et post-conditions et les spécifications algébriques.

Exemple, pour la spécification suivante des piles d'entiers :

```
type
    stack-of-int
sorts
    stk
constructors
    New : -> stk
    Push : (int,stk) -> stk
operators
    Pop: stk -> stk
    Top: stk -> int
exceptions
    Undef-stk
    Undef-int
variables
    n : int
    s : stk
equations
  1:  Pop(Push(n,s)) ==> s
  2:  Top(Push(n,s)) ==> n
conditions
  1:  Pop(New) raises Undef-stk
  2:  Top(New) raises Undef-int
end stack-of-int
```

Une implémentation I pour une spécification algébrique (hiérarchique, dans le cadre présenté par Huimin Lin) consiste en :

- un type de représentation T_s avec un invariant Inv^I et une relation d'équivalence $=_S^I$ sur T_s pour la sorte s de la spécification,
- une procédure f^I pour chaque fonction $f : (s_1, \ldots, s_n) \text{ -> } s$.
  f^I a soit n paramètres d'entrée de sortes $s_1 \ldots s_n$ et un paramètre résultat de sorte s soit $n-1$ paramètres en entrée de sortes $s_1 \ldots s_{n-1}$ et $s_n = s$ en entrée-sortie.

Nous pouvons avoir l'implémentation suivante, où la sorte *stk* est codée par une paire "tableau et indice", dont la taille maximum du tableau est 100 (invariant) :

$$T_{stk} \equiv \left\{ \begin{array}{l} a : \texttt{array[1..100] of integer} \\ p : \texttt{integer} \end{array} \right.$$

ainsi que les corps de programmes suivants (op^{SI} est l'équivalent de l'opérateur *op* dans l'implémentation *SI* de *Stack-of-int*) :

$New^{SI}(result\ s) \equiv s.p := 0$
$Pop^{SI}(value\text{-}result\ s) \equiv \mathit{if}\ s.p > 0\ \mathit{then}\ s.p := s.p - 1\ \mathit{endif}$

$Top^{SI}(value\ s, result\ m) \equiv if\ s.p > 0\ then\ m := s.a[s.p]\ endif$
$Push^{SI}(value\ n, value-result\ s) \equiv if\ s.p < 100\ then\ s.a[s.p+1], s.p := n, s.p+1\ endif$

La notation $x, y := a, b$ est l'affectation parallèle[2] de a à x et de b à y.

Huimin Lin propose une méthode de dérivation de tout terme construit à partir des opérateurs de *Stack-of-int*. Cette translation $(\_)^I$ est définie récursivement sur la structure des Σ-termes.

À chaque terme t est associée une variable auxilliaire r_t qui désigne la variable qui possède la valeur de t dans le code qui en est dérivé. Une substitution θ_t est introduite pendant la transformation de t. Appliquée à r_t, nous obtenons la variable effective à laquelle r_t se réfère (par exemple, si r_t est le paramètre en mode value-result, $r_t\theta_t$ désigne la variable qui se trouve à la position du paramètre r_t. Soit t' l'opérande dans t qui se trouve à la position de r_t, alors θ_t est telle que $r_t\theta_t = r_{t'}\theta_{t'}$).

Une implémentation I est correcte (la transformation $(\_)^I$ est définie pour transformer les termes en programmes) si toutes les équations sont satisfaites par l'implémentation. Une équation $\forall X \cdot t = t'$ est satisfaite par l'implémentation lorsque pour une exécution de t^I et de t'^I avec des valeurs initiales identiques pour les variables de X, les valeurs après exécution sont identiques dans les deux cas. Ceci se traduit en termes de plus faibles préconditions par l'équivalence des plus faibles préconditions des termes dérivés des parties droite et gauche de chaque équation, lorsque les conditions de leurs terminaisons sont égales et non fausses (valeur du prédicat T de terminaison différente de "faux"),

Pour être générale, la notion de d'implémentation correcte est paramétrée par un ensemble de termes clos appelés "observations". Ces termes doivent être conservés différents tant qu'ils ne sont pas forcés à être égaux par les axiomes (équations qui définissent les opérateurs). Pour la sémantique initiale, ce sont tous les termes fermés, pour la sémantique hiérarchique, tous les termes des sortes de base.

Une implémentation préserve le comportement de la spécification SP respectivement à l'ensemble d'observations OBS si :

$$\forall t, t' \in OBS, I \models t \neq t' \text{ à chaque fois que } SP \not\vdash t = t'$$

Limitations de la technique de dérivation d'un terme dans une implémentation de Huimin Lin :

La règle de dérivation d'un terme t dont un opérateur défini est en racine est la suivante : soit $t = f(x_1, \ldots, x_{i_1-1}, t_{i_1}, x_{i_1+1}, \ldots, x_{i_k-1}, t_{i_k}, x_{i_k+1}, \ldots, x_n)$, $0 \leq i_1 \leq \cdots \leq i_k \leq n$, $k \geq 0$ où t_{i_j} ne sont ni des variables, ni des constantes, la dérivation dans le cas où f est un opérateur défini est la suivante :

$$\begin{aligned} t^I = \quad & t^I_{i_1}; \ldots; t^I_{i_j}; \ldots; t^I_{i_k}; \\ & f^I(x_1, \ldots, r_{t_{i_1}}\theta_{t_{i_1}}, \ldots, r_{t_{i_k}}\theta_{t_{i_k}}, \ldots, x_{n'-1}, x_{n'}); \end{aligned}$$

$$\text{et } \theta_t = \begin{cases} [r_t \mapsto x_{n'}] & \text{si } i_k < n' \\ [r_t \mapsto r_{t_{i_k}}]\theta_{t_{i_k}} & \text{si } i_k = n' \end{cases}$$

2. Cette affectation permet de s'affranchir des conflits qui peuvent apparaître lorsque des variables sont à la fois en partie droite et en partie gauche de l'affectation. Elle n'existe pas en Ada, langage choisi pour notre domaine concret, et la méthode que nous proposons résout ces conflits.

Le premier cas du calcul de θ_t correspond à une procédure où le paramètre formel de retour de f^I est un paramètre en sortie uniquement, dont le paramètre effectif est donc une variable "fraîche". Le second cas correspond à une procédure où le paramètre formel de retour est un paramètre en entrée-sortie et dont le paramètre effectif est la variable $r_{t_{i_k}}\theta_{t_{i_k}}$ qui contient la valeur de l'opérande à cet emplacement. La valeur n' est soit $n+1$ dans le premier cas, soit n dans le second.

Cette technique n'est pas correcte lorsque plus d'un terme t_{i_j} est implémenté par une procédure qui modifie son dernier paramètre, et que le paramètre effectif utilisé à cette position pour les différents t_{i_j} est une unique variable. Trivialement, cela revient à traduire le terme $append(reverse(x), reverse(x))$, avec la sémantique habituelle des opérateurs utilisés, en :

$reverse^I(x);\ reverse^I(x);$
$append^I(x,x);$

avec le résultat dans x. Evidemment, le résultat n'est pas le même dans le programme impératif que dans le terme fonctionnel.

Par rapport à [Lin93], les avantages de notre approche, en dehors de l'aspect d'optimisation des duplications sont :

- notre lien d'implémentation est un objet formel qui décrit comment un constructeur ou opérateur quelconque peut être interprété dans le domaine concret, pour une représentation de ses opérandes,
- les opérateurs et constructeurs ne sont pas limités à un co-domaine mono-sorte,
- la correspondance des paramètres n'est pas obligatoirement une même position pour un même opérande,
- les sortes du co-domaine peuvent correspondre à des paramètres formels autres que celui en dernière position (ce qui ne restreint pas l'opérande modifié à celui qui est placé en dernier) et n'impose pas la dernière sorte du domaine égale à celle du co-domaine,
- les constructeurs et opérateurs définis peuvent indifféremment être une fonction ou une procédure.

■

Approche de Nancy Zambrano

Nancy Zambrano propose dans [Zam95] une dérivation des spécification algébriques vers des programmes Ada. L'axiomatisation des opérateurs est définie par des pré- et post-conditions. Pour les pré-conditions dont les assertions sont vraies, les post-conditions définissent la valeur du résultat de l'application de l'opérateur concerné sur ses opérandes. Des assertions semblables définissent le domaine d'application de l'opérateur lorsque celui-ci est partiel.

Les opérateurs sont de co-domaine mono-sorte (pas de produit cartésien de sortes) et peuvent être implémentés par des fonctions ou procédures. Le paramètre formel en sortie d'une procédure, qui correspond au résultat de son application, n'a pas de position fixée.

La dérivation se fait en deux étapes :

La première étape consiste en l'obtention d'une "*spécification algébrique opérationnelle*" qui combine pour les sortes et opérateurs dérivés, les types et interfaces Ada correspondants, et des assertions sur les paramètre formels des fonctions ou procédures et sur le résultat dans le cas d'une fonction.

Exemple 2.1: Pour un opérateur *pop* de dépilement de pile, dont l'unité algébrique est implémentée par un paquetage basé sur les listes :

```
procedure POP(mod S: STACK);
  pre:  is-empty-list(|s|) is-false
  post: s' = <tail(|s|)>
```

■

Cette spécification algébrique opérationnelle est obtenue à partir de :

- la spécification algébrique d'origine à dériver,
- la "spécification de référence" qui est l'axiomatisation de tous les opérateurs utilisés dans la spécification algébrique d'origine,
- la "signature opérationnelle" (interface Ada avec les paramètres qui correspondent) fournie pour la spécification algébrique d'origine,
- une "application de modélisation" R qui va des sortes et opérations de la signature opérationnelle vers les sortes et opérateurs de la spécification de référence.

La seconde étape de dérivation d'une spécification algébrique opérationnelle vers un paquetage Ada (implémentation concrète) est décrite en partie formellement : elle consiste en une traduction du test que forme chaque précondition, suivi de la traduction du calcul de la post-condition.

Définition : un *terme simple* est un terme qui ne contient pas de "*if-then-else*". Le terme pré-condition est un terme simple (de sorte booléenne). Le terme post-condition est de la forme obtenue par application de la règle de distributivité du *if-then-else* tant que cela est possible: un sous-terme *if-then-else* n'est jamais en position d'opérande d'un terme qui n'est pas un *if-then-else*.

Les différents cas où une nouvelle variable locale doit être introduite (conflit d'accès à une variable, incompatibilité d'utilisation d'un paramètre formel avec son mode, ...) sont énumérés.

La dérivation d'un terme s'effectue classiquement en traduisant récursivement les *if-then-else* imbriqués, et pour chaque terme condition d'un *if-then-else* ou sous-terme *then* ou *else* qui est un terme simple, en appliquant récursivement la traduction de

chaque opérateur sur la dérivation de ses opérandes. Il n'y a pas de problème de destination commune à un *if-then-else* puisque le résultat de l'évaluation d'un *if-then-else* n'est jamais utilisé dans un calcul. En fin de la dérivation de la post-condition, les "`return`" ou les affectations aux paramètres de retour sont insérés aux endroits adéquats.

La méthode de dérivation proposée par Nancy Zambrano peut être améliorée sur différents points :

- l'application R qui correspond à notre lien d'implémentation ne décrit pas explicitement la correspondance des paramètres,
- les conflits d'utilisation de variables sont définis par une utilisation d'une variable en tant que paramètre modifié alors qu'il y a plus d'une occurrence de la variable dans le terme à dériver. Ils sont résolus systématiquement en introduisant une nouvelle variable locale où est recopiée la valeur de la variable conflictuelle, et qui est utilisée à sa place comme paramètre modifié.

  Le fait que seul un sous-terme *then* ou *else* d'un *if-then-else* est exécuté pendant une évaluation de la post-condition, fait que les variables locales d'un sous-terme *then* sont réutilisée comme variables locales du sous-terme *else*. Par contre cette propriété n'est pas utilisée pour éviter de calculer des conflits entre les sous-termes *then* et *else* d'un *if-then-else*.

Nancy Zambrano propose une extension de sa méthode pour traiter des opérateurs dont le co-domaine est un produit cartésien de sortes, et qui sont donc implémentés par des procédures avec plusieurs paramètres formels en entrée-sortie. Cette extension consiste en une séparation du calcul du résultat en des calculs disjoints de chaque élément du résultat (séparation de la post-condition en plusieurs post-conditions). Les textes impératifs sont dérivés séparément comme il est décrit dans la thèse, et sont assemblés séquentiellement pour être exécutés l'un après l'autre. Des duplications de variables sont ajoutées dès qu'un texte dérivé modifie une variable utilisée par un autre texte.

La séparation en plusieurs sous-calculs disjoints impose de dupliquer les calculs communs, par exemple si l'on possède l'opérateur "*let* $v = t_1$ *in* t_2" qui définit une variable locale v dont la valeur est calculée par t_1, et qui est utilisée dans t_2, et dont le résultat, t_2 est de sorte produit cartésien de sortes.

Cette méthode effectue dans certains cas des duplications, alors qu'un ordre adéquat entre textes dérivés les aurait évitées. Lors d'un conflit d'utilisation d'un paramètre formel, une duplication est effectuée et la variable supplémentaire qui contient la valeur dupliquée est utilisée comme paramètre effectif en entrée-sortie. Il peut être économe de modifier directement le paramètre formel (et utiliser la duplication par les autres textes conflictuels) lorsqu'il est en mode entrée-sortie et doit contenir précisément ce résultat. Cette méthode ne traite pas les conflits qu'il peut y avoir pour les "affectations finales" (lorsqu'elles correspondent à une affectation parallèle, par exemple : $x, y := v_1, x$). La procédure SWAP définie de la manière suivante sera implémentée par un code incorrect si l'on ne tient pas compte des conflits dans les affectations supplémentaires :

```
procedure SWAP(mod X,Y: NAT);
post:  x' = y  ∧  y' = x
```

La méthode que nous proposons résout tous ces cas en ordonnant l'ensemble des calculs de manière à minimiser les duplications de sauvegarde.

■

Autres cas de dérivation

Bernd Krieg-Brückner décrit dans [KB86] comment passer d'une interface fonctionnelle d'un programme Ada à une interface procédurale. Pour la transformation du corps d'une fonction, autant de variables auxiliaires sont introduites qu'il y a d'appels imbriqués de fonctions. Ces appels sont transformés en appels de procédures, mais les programmes obtenus sont incorrects en cas de conflits d'accès aux paramètres formels du programme.

Le travail décrit dans [BA90], effectué dans le cadre du projet PROPOS (PROgrammation par comPOSants) par Didier Bert et Cyril Autant est le point de départ de la méthode de dérivation que nous présentons. Il consiste en une définition d'un cadre formel des dérivations de spécifications vers des programmes. Il précise en particulier ce que l'on entend par domaine abstrait (celui des spécifications) et par domaine concret (celui des programmes). Les langages des domaines abstraits et concrets choisis sont, respectivement LPG, et Ada.

Trois formes de dérivation sont proposées :

- dérivation fonctionnelle,
- dérivation orientée "type abstrait",
- dérivation vers des modules encapsulés.

Pour chaque cas, le formalisme du domaine concret est donné, ainsi que le lien entre la signature algébrique et l'interface Ada. Une axiomatisation des programmes est décrite, qui en s'appuyant sur la logique de Hoare, permet de prouver qu'un programme respecte des assertions sur ses entrées-sorties. Une fonction d'interprétation est définie des objets du domaine concret vers ceux du domaine abstrait (la signature algébrique). Une fonction δ de dérivation de l'interface Ada à partir de la signature algébrique est définie. La démarche de dérivation $\delta^{\#}$ des corps de programmes Ada est finalement décrite pour les trois formes de dérivation. Elle consiste en :

- mise sous forme fonctionnelle,
- séquentialisation des appels,
- détermination des dépendances de variables :
  - déterminer les conflicts,
  - déterminer les copies à réaliser,
  - générer les calculs.

Nous avons choisi dans notre thèse de résoudre le problème de dérivation de corps de programmes dans un cas qui s'apparente à la dérivation "type abstrait". Nous n'avons pas repris le codage du lien entre une signature algébrique et une interface Ada car il imposerait une manipulation lourde d'indices (correspondance de paramètres). Nous lui avons préféré un codage en un seul objet fonctionnel (un combinateur) du "passage de paramètre à l'appel" ou de la "récupération de paramètre en retour".

Chapitre 3

Une méthode de dérivation de spécifications algébriques vers des programmes impératifs : cadre, problématique et principe

3.1 Cadre et problématique

3.1.1 Système favorisant la réutilisation

Nous proposons de définir un système de transformation de spécifications algébriques en un langage impératif: utilisant à la fois le style fonctionnel et le style procédural.

Le développement se fait ainsi au niveau des spécifications algébriques, en bénéficiant de la clarté et de la sûreté de la programmation fonctionnelle et de la sémantique algébrique. L'efficacité n'est pas apportée par le langage de bas niveau seul, mais par les possibilités d'optimisation offertes par une description initiale abstraite et un niveau de réalisation où les défauts des langages fonctionnels (par exemple la gestion de la mémoire) peuvent être supprimés.

Les bibliothèques algébriques existantes peuvent avoir plus d'une implémentation impérative. La dérivation d'une spécification utilisant une bibliothèque doit donc aboutir à un programme utilisant toute l'implémentation de cette bibliothèque. Cette propriété apporte plusieurs avantages :

- dériver toute spécification de manière à la tester dans un contexte précis : station ou processeur dédié, interface homme/machine différente, environnement ou système d'exploitation particulier, ...
- réutiliser toute spécification par son intégration dans la bibliothèque algébrique, soit par sa dérivation automatique, soit en fournissant sa réalisation concrète et ce qui les lie.
- intégrer un programme impératif à la bibliothèque, et offrir ainsi son utilisation à partir d'un langage de haut niveau. Ceci est possible dès lors que l'on sait lui associer sa spécification algébrique.

Les systèmes formels choisis sont, d'une part, le langage LPG pour les spécifications algébriques d'origine (mais ce pourrait être tout autre langage de spécification algébrique), et d'autre part le langage Ada pour le langage de programmation.

Ce système doit donc être capable de passer d'un style purement fonctionnel : LPG, à un style impératif (où des modifications "en-place" de variables peuvent être faites par des affectations ou des appels de procédures) : Ada, et permettre ainsi de réutiliser des bibliothèques de spécifications algébriques dont les implémentations en Ada, validées, imposent les interfaces (ordre et modes fixés des paramètres des fonctions ou procédures).

Le schéma général est décrit à la figure 3.1 suivante.

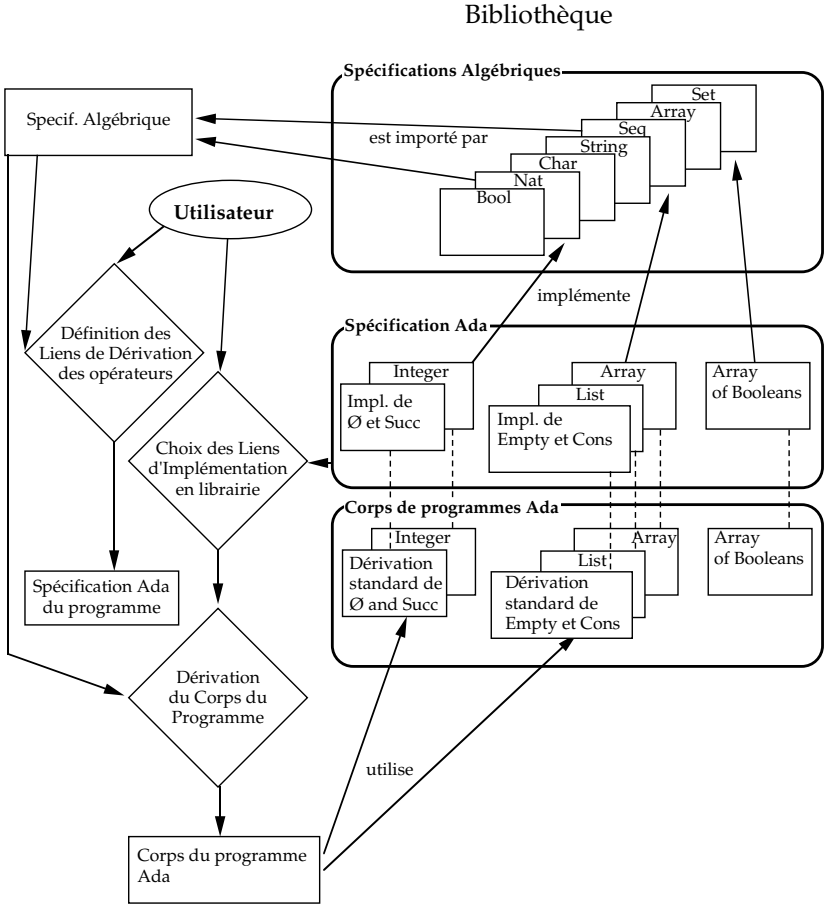


FIG. 3.1 – *Système d'implémentation de Spécifications Algébriques avec Bibliothèque*

Dans ce système, l'utilisateur qui veut transformer en Ada une spécification doit choisir pour chacun des opérateurs de cette spécification son *style* : s'il souhaite obtenir une fonction ou une procédure, son *profil* : le type et le mode (in, out, in out) de chacun de ses paramètres formels, et la *correspondance des paramètres* entre ceux de l'opérateur algébrique et ceux de la fonction ou procédure dérivée.

Exemple 3.1: Ayant la spécification de l'opérateur algébrique `sortnat : seqnat ->seqnat` de tri de séquence d'entiers, et en sachant que la sorte

`seqnat` est implémentée par le type `listnat`, nous devons indiquer pour une fonction ou procédure de nom `trier` que nous souhaitons dériver, la correspondance de chaque paramètre :

1. `function trier(l: in listnat) return listnat`, l'opérande de `sortnat` correspond à `l` et le résultat algébrique au résultat de l'évaluation de la fonction.
2. `procedure trier(l: in listnat; ltriee: out listnat)`, l'opérande de `sortnat` correspond au premier paramètre `l`, et le résultat algébrique au deuxième paramètre `ltriee` en retour de l'exécution de la procédure.
3. `procedure trier(ltriee: out listnat; l: in listnat)`, l'opérande de `sortnat` correspond au deuxième paramètre `l`, et le résultat algébrique au premier paramètre `ltriee` en retour de l'exécution de la procédure.
4. `procedure trier(l: in out listnat)`, l'opérande de `sortnat` correspond à l'unique paramètre `l`, et le résultat algébrique au même paramètre `l` modifié en retour de l'exécution de la procédure.

■

Le nombre de possibilités augmente avec l'arité de l'opérateur, et particulièrement si nous considérons les opérateurs dont le co-domaine est un produit cartésien de sortes : chaque sorte du co-domaine, si elle existe dans le domaine, peut correspondre à un paramètre `in out` dans le cas d'une dérivation vers une procédure. Ceci est montré dans l'exemple suivant :

Exemple 3.2: Soit l'opérateur suivant :

```
insdico : (string,seq[string],dico[string])
          ->(dico[string],boolean,seq[string])
```

un opérateur d'insertion d'un mot et de sa définition dans un dictionnaire, tel que : $insdico(m, l_1, d_1) = (d_2, e, l_2)$ avec m un mot, l_1 sa définition, d_1 un dictionnaire, et comme résultat d_2 égal à d_1 augmenté du mot m et de sa définition, e un booléen égal à $vrai$ si m avait déjà une définition dans d_1, et l_2 sa définition précédente si elle existait.

Cet opérateur peut avoir comme implémentation, avec une correspondance évidente :

1. ```
   procedure dicinsert(cle: in str20;
                       new_def: in liste_str20;
                       dico: in str_strlist_map;
                       new_dico: out str_strlist_map;
                       existe: out boolean;
                       anc_def: out liste_str20 )
   ```
2. ```
   procedure dicinsert(cle: in str20;
                       new_def: in liste_str20;
                       dico: in out str_strlist_map;
   ```

```
                        existe: out boolean;
                        anc_def: out liste_str20)


3. procedure dicinsert(cle: in str20;
                        def: in out liste_str20;
                        dico: in out str_strlist_map;
                        existe: out boolean)


4. ...
```

■

De nombreuses conséquences découlent du choix du style et du profil. La première est la forme de l'appel: soit $t = g(t_1, \ldots, op(\ldots), \ldots, t_n)$ selon que op est dérivé en une fonction `F` ou une procédure `P`, il pourra être en position de paramètre effectif dans le code dérivé de t ou devra précéder ce code. Dans ce cas, ce sera une variable (ou une expression-gauche) en paramètre effectif de sortie de `P` qui sera paramètre effectif d'entrée ou entrée-sortie du code dérivé de g. Une seconde conséquence qui rend plus complexe la dérivation d'un texte impératif est la destruction de la valeur d'une variable en paramètre effectif d'un paramètre en mode entrée-sortie. Ces cas seront détaillées dans la section 3.1.3 traitant de la problématique.

3.1.2 Lien d'implémentation

Nous allons maintenant décrire le lien entre les composants d'une spécification algébrique et ceux d'un paquetage Ada. Chaque module de la bibliothèque algébrique peut avoir plusieurs implémentations en Ada. Nous voyons donc dans la figure 3.1 de nombreuses flèches de nom *implémente*, ce qui signifie que le paquetage en origine *implémente* ou *interprète* dans le domaine concret l'unité algébrique désignée.

La dérivation d'une spécification algébrique vers Ada se fera pour une implémentation fixée de chaque bibliothèque algébrique importée. Le lien entre un module LPG et un module Ada consiste en une relation Δ entre la signature LPG et l'interface Ada.

Cette relation Δ est une application qui, à partir de la signature d'une spécification algébrique LPG, permet d'obtenir l'interface Ada du paquetage qui implémente cette spécification algébrique. Elle est nécessaire pour calculer la fonction d'interprétation (ou d'abstraction) $\mathcal{I}$, ainsi que la fonction de dérivation (ou de concrétisation) $\Delta^{\#}$, inverse de $\mathcal{I}$. La fonction Δ existe pour tous les modules algébriques de la bibliothèque, et doit être définie par l'utilisateur pour la spécification algébrique qu'il souhaite dériver.

La fonction $\Delta^{\#}$ de dérivation de l'axiomatisation d'un opérateur algébrique en corps de programme Ada est une extension de la fonction Δ à l'axiomatisation des opérateurs

définis. Son application est décrite à la figure 3.2 suivante :

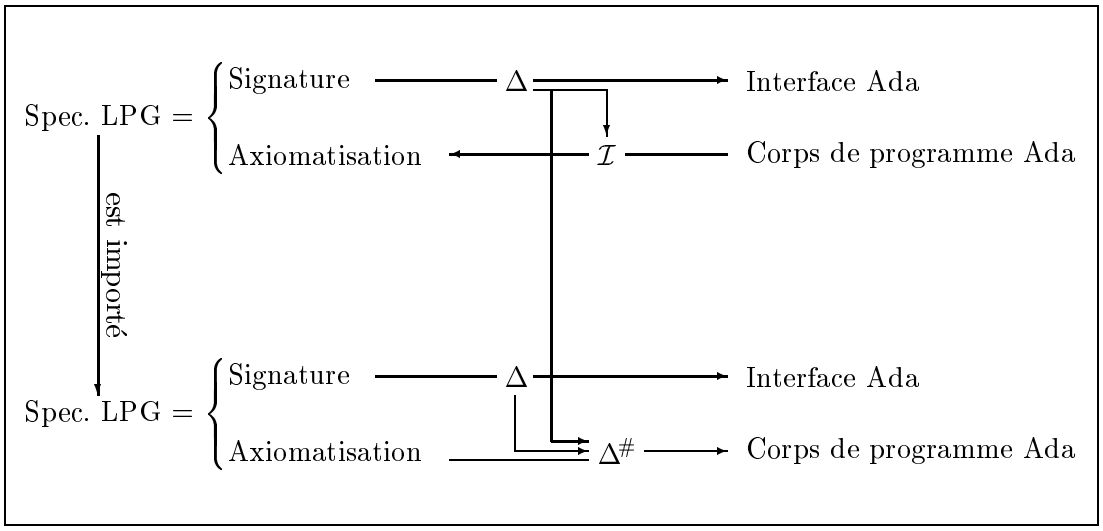


FIG. 3.2 – *Dérivation d'une Spécifications Algébrique avec Bibliothèque*

Nous allons décrire précisément les différents composants de Δ, qui n'est pas un simple morphisme de signature puisque pour des opérateurs implémentés par des procédures, les résultats peuvent apparaître comme paramètres supplémentaires (en mode `out`), ou bien être "repliés" sur des paramètres existants qui deviennent de mode `in-out`. En effet, une correspondance de noms ne suffit pas.

Définition 3.1: Soit SP une spécification algébrique, $SP = (\Sigma, E)$. La signature $\Sigma = (S, \Omega)$ est composée de l'ensemble S des sortes et $\Omega = C \cup D$ l'union des constructeurs C et des opérateurs définis D. L'ensemble E d'équations orientées constitue l'axiomatisation des opérateurs définis de D.

Soit Δ_{SP} le lien d'implémentation choisi pour SP (ou lien de dérivation défini pour SP).

$$\Delta_{SP} = (\Delta_\Sigma, \Delta^{\#})$$

L'application de dérivation $\Delta^{\#}$ qui permet de dériver automatiquement les corps de programmes impératifs des fonctions ou procédures qui implémentent les opérateurs définis (lorsque l'on souhaite faire une dérivation) est décrite à la section 3.2.2 page 63. L'application Δ_Σ a pour domaine la signature Σ, et pour co-domaine une interface Ada. L'obtention de cette interface est décrite dans la section 3.2.1 page 60.

$\Delta_\Sigma = (\Delta_S, \Delta_C, \Delta_D)$ avec $\Delta_S, \Delta_C, \Delta_D$ les fonctions annexes suivantes :

$\Delta_S : S \text{ -> } \mathcal{T}$ est une fonction des sortes vers les types. Elle associe par exemple la sorte `nat` au type `integer` prédéfini.

$\Delta_C : C \text{ -> } (\mathcal{E}_C, In_C, Out_C, Is_C, Select_C)$ est une application qui associe à un constructeur $c \in C$ une description de son implémentation en Ada et des informations nécessaires à son utilisation, par un 5-uplet :

$$\forall c \in C \ \cdot \ \Delta_C : c \mapsto (\mathcal{E}_C(c), In_C(c), Out_C(c), Is_C(c), Select_C(c))$$

Lorqu'il n'y a pas d'ambiguïté sur les ensembles d'opérateurs (constructeurs, ou définis comme plus loin), nous omettons de les préciser pour les applications $\mathcal{E}, In, Out, Is, Select$.

$\mathcal{E}(c)$ est texte impératif avec variables liées universellement dont l'exécution réalise l'application du constructeur c en LPG. Ce peut être un appel de fonction ou de procédure, une séquence d'instructions ou une expression fonctionnelle Ada. Par exemple l'opérateur standard *cons* pourrait avoir pour lien d'implémentation dans le type liste :

$$\mathcal{E}(cons) = [x_1, x_2]\{\texttt{new liste'(elem =>}x_1\texttt{; succ =>}x_2\texttt{)}\}$$

$In(c)$ et **$Out(c)$** définissent comment se fait le passage de paramètres, et où se trouve le résultat en fin d'exécution de $\mathcal{E}(c)$: $In(c)$ est une correspondance des paramètres de la fonction ou procédure ou des variables du texte Ada, aux représentations des opérandes du constructeur. Out est une correspondance des paramètres de la procédures ou des variables du texte Ada, aux termes LPG dont ils doivent posséder la valeur après exécution.
Les variables liées $x_1 \dots x_p$ de $\mathcal{E}(c) = [x_1, \dots, x_p]\{\cdots\}$ dites d'*entrée*, qui correspondent aux opérandes de c, sont alors obtenues par $In(c)^{-1}(x_1, \dots, x_p)$ et celles dites *de sortie*, qui correspondent aux résultats de l'application de c, par $Out(c)^{-1}(x_1, \dots, x_p)$.

Le testeur $Is(c)$ est une fonction Ada booléenne qui, appliqué à une expression Ada, renvoie la valeur *vrai* si celle-ci est une représentation d'un terme LPG construit par c. Nous utiliserons une notation curryfiée : $is^c(x)$ pour $Is(c)(x)$.

Le sélecteur $Select(c)$ est une fonction qui, pour un entier i et une représentation Ada d'un terme LPG t construit par c, renvoie une représentation du $i^{ème}$ sous-terme de profondeur 1 de t. Nous utiliserons une notation curryfiée : $select_i^c(x)$, $i \in [1 \dots Card(Dom(c))]$ pour $Select(c)(i, x)$.

$\Delta_D : D \text{ -> } (\mathcal{E}, In, Out)$ est une fonction qui associe à un opérateur défini d une description de son implémentation en Ada et des informations nécessaires à son utilisation. Pour $d \in D$, $In(d)$ et $Out(d)$ sont les mêmes que pour un constructeur. Pour intégrer à la bibliothèque une spécification et son implémentation, $\mathcal{E}$ peut être un texte quelconque, mais pour effectuer une dérivation, $\mathcal{E}$ doit être uniquement la forme d'appel d'une procédure ou fonction. $\mathcal{E}, In$ et Out sont en effet nécessaires pour dériver une interface Ada d'une signature algébrique, ou le code d'une procédure ou fonction à partir du terme en partie droite de l'équation qui définit un opérateur.

■

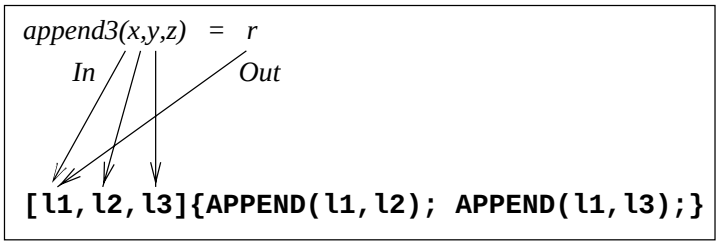
Le texte d'implémentation Ada $\mathcal{E}(c)$ d'un constructeur ou opérateur défini doit respecter les mêmes conventions que le corps d'une procédure Ada vis-à-vis de ses paramètres : il ne doit pas modifier les variables liées qui sont en entrée seulement, et de ne pas lire la valeur de celles qui sont en sortie uniquement.

Dans les exemples suivants, les paramètres d'une procédure appelée qui sont modifiés sont soulignés. Nous observons dans l'exemple 3.3 que la variable liée modifiée est celle qui est en entrée et en sortie. Par contre, si dans l'exemple 3.4, une représentation du résultat algébrique est bien dans la variable liée de sortie, une variable d'entrée uniquement a été modifiée.

Exemple 3.3: Soit $append3 : (Seq[elem], Seq[elem], Seq[elem]) \rightarrow Seq[elem]$ un opérateur algébrique qui effectue la concaténation de trois listes. Il peut être implémenté en utilisant une procédure Ada `APPEND(la: in out liste; lb: in liste)` qui concatène ses deux paramètres et retourne le résultat dans le premier. Pour cela, le texte Ada de $\mathcal{E}(append3)$ contient deux appels consécutifs à `APPEND` :

$$\mathcal{E}(append3) = [l_1, l_2, l_3]\{\texttt{APPEND}(\underline{l_1}, l_2); \texttt{APPEND}(\underline{l_1}, l_3);\}$$

avec en entrée l_1, l_2, l_3 et en sortie l_1. Ce qui donne avec notre notation, $In(append3)^{-1}(l_1, l_2, l_3) = (l_1, l_2, l_3)$, et $Out(append3)^{-1}(l_1, l_2, l_3) = (l_1)$:



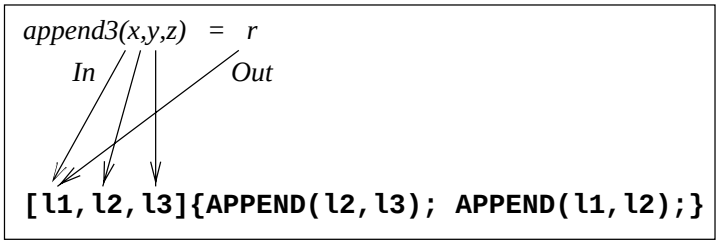
Nous vérifions que l_1 qui est la seule variable modifiée par l'exécution de $\mathcal{E}(append3)$ est la seule à la fois en entrée et en sortie.

■

Exemple 3.4: En reprenant le même opérateur $append3$ et la même procédure `APPEND` :

$$\mathcal{E}(append3) = [l_1, l_2, l_3]\{\texttt{APPEND}(\underline{l_2}, l_3); \texttt{APPEND}(\underline{l_1}, l_2);\}$$

avec $In(append3)^{-1}(l_1, l_2, l_3) = (l_1, l_2, l_3)$, et $Out(append3)^{-1}(l_1, l_2, l_3) = (l_1)$:



Le résultat correct est bien dans l_1 après l'exécution de $\mathcal{E}(append3)$, mais la modification de l_2 ne respecte pas les modes (entrée, sortie, entrée-sortie) des variables liées de $\mathcal{E}$, c'est-à-dire seul l_1 modifié puisque :

$$l_1 \in In(append3)^{-1}(l_1, l_2, l_3) \cap Out(append3)^{-1}(l_1, l_2, l_3)$$

■

La définition formelle du lien Δ et de sa mise en œuvre est faite à la section 3.1.5 page 53.

3.1.3 Problématique liée à l'introduction de procédures

La dérivation de la partie gauche de l'unique équation définissant l'axiomatisation d'un opérateur c donne l'en-tête de la fonction ou procédure l'implémentant, ainsi que, dans le cas d'une procédure, les variables qui sont les paramètres formels destinations des représentations du (ou des) résultat(s) algébrique(s). La dérivation du terme T en partie droite doit donner le corps de cette fonction ou procédure. Nous appelons "opérateur d'intérêt" l'opérateur dont nous dérivons l'axiomatisation.

Soit $\Delta^{\#}$ une fonction d'implémentation de LPG vers Ada. Si l'on reprend le premier exemple de la section 3.1.1 sur la réutilisation, la traduction des équations de `sortnat` doit tenir compte de certaines caractéristiques des choix de dérivation et des bibliothèques. Les différents points à traiter sont les suivants :

Forme de l'appel, passage des paramètres effectifs et résultat : L'appel de la fonction ou procédure `trier` est différent selon sa nature, l'ordre et le mode de ses paramètres formels. Nous choisirons pour simplifier des implémentations fonctionnelles des opérandes de l'opérateur principal. Ce qui donne pour les différentes implémentations de `sortnat` de l'exemple 3.1 page 42 :

1. $\forall t : seqnat \cdot \Delta^{\#}(sortnat(t)) =$ `trier(`$\Delta^{\#}$`(t))`
2. $\forall t : seqnat \cdot \Delta^{\#}(sortnat(t)) =$ `trier(`$\Delta^{\#}$`(t),z)` résultat dans z
3. $\forall t : seqnat \cdot \Delta^{\#}(sortnat(t)) =$ `trier(z,`$\Delta^{\#}$`(t))` résultat dans z
4. $\forall x : seqnat \cdot \Delta^{\#}(sortnat(t)) =$ `z := `$\Delta^{\#}$`(t); trier(z)` résultat dans z

Forme impérative interdite en position d'opérande : Le texte dérivé d'un sous-terme t d'un terme T ne peut pas être placé en position de paramètre effectif s'il n'est pas de forme fonctionnelle, c'est-à-dire s'il est dérivé en un appel de procédure, une affectation, un "*if-then-else*", une composition de ces instructions, etc. Il est alors nécessaire de créer une variable intermédiaire :

> Soit $\Delta^{\#}$ la première des implémentations précédentes, où `sortnat` est en forme procédurale. Soit z une nouvelle variable libre, et `op` un opérateur dont l'implémentation est une fonction unaire `foo`. Nous avons : $\Delta^{\#}(op(sortnat(x)))$ = { `trier(x,z) ;` } résultat : `foo(z)`

Nous voyons que le code dérivé de tout terme comporte une partie procédurale, dès lors qu'un opérateur ou constructeur de ce terme est implémenté par une forme procédurale (appel de procédure ou affectation).

Forme d'opérande interdite en mode `in-out` : Le terme algébrique dont l'opérateur principal devient une procédure avec un paramètre formel en mode `in-out` ne peut avoir comme paramètre effectif un appel de fonction : celui-ci doit être une composition de sélecteurs primitifs du langages sur une donnée mutable : une *expression-gauche*. Sinon une variable intermédiaire doit être introduite :

$\Delta^{\#}(sortnat(tail(l)))$ peut être dérivé en :

- soit `trier(`$\Delta^{\#}(l)$`.suivant)`
  dans le cas où la queue de la liste n'est pas utilisée
  ultérieurement et qu'une modification "en-place" est autorisée,
  si $\mathcal{E}(tail) = [v]\{$`v.suivant`$\}$ est une expression-gauche,
  $In(tail)^{-1}(v) = (v)$ et $Out(tail)^{-1} = < \phi >$
- soit `w := cdr(`$\Delta^{\#}(l)$`); trier(w)`
  si $\mathcal{E}(tail) = [v]\{$`cdr(v)`$\}$ est un appel de fonction,
  $In(tail)^{-1}(v) = (v)$ et $Out(tail)^{-1} = < \phi >$

Ecriture interdite d'un paramètre formel en mode `in` uniquement :
Une variable qui correspond à un paramètre formel de l'opérateur dérivé en mode `in` uniquement doit être dupliquée et sa duplication doit être utilisée à sa place en paramètre effectif en mode `out` ou `in-out` dans un appel de procédure ou dans une instruction d'affectation.

Lecture interdite d'un paramètre formel en mode `out` uniquement :
Une variable qui correspond à un paramètre formel de l'opérateur dérivé en mode `out` uniquement doit être utilisé uniquement en position gauche d'une instruction d'affectation ou en paramètre effectif en mode `out` uniquement d'un appel de procédure.

Nous venons de voir les différentes caractéristiques et contraintes de l'utilisation d'un langage impératif pour implémenter un opérateur algébrique. Nous devons en tenir compte pour obtenir un code à partir de son axiomatisation.

3.1.4 Spécification source

Nous avons choisi de dériver des spécifications algébriques d'un sous-langage du langage de spécification LPG (langage de programmation générique) comprenant :

- Un ensemble S de sortes,
- Un ensemble de constructeurs C, dont certains sont prédéfinis, et d'autres définis par l'utilisateur,
- Un ensemble d'opérateurs définis D dont l'axiomatisation est définie par un ensemble d'équations,

- Un ensemble A d'équations orientées définissant l'axiomatisation des opérateurs définis.

Particularité importante du langage LPG : le co-domaine d'un opérateur défini peut être un produit cartésien de sortes. Exemple : soient deux opérateurs :

$op : (s_1, \ldots, s_n) \text{ -> } (s_{n+1}, \ldots, s_{n+m})$ et $op' : (s_{n+1}, \ldots, s_{n+m}) \text{ -> } (s'_1, \ldots, s'_p)$
et n termes $t_1, \ldots, t_n$ de sortes respectives $s_1, \ldots, s_n$ alors le terme $op'(op(t_1, \ldots, t_n))$ est correct et a pour valeur un produit cartésien de p valeurs de sortes respectives $s'_1, \ldots, s'_p$. Les opérateurs dont le co-domaine est un produit cartésien de sortes sont implémentés par des procédures.

Les spécifications d'origine que nous dérivons sont :

- soit un type comportant :
  - une sorte,
  - des constructeurs,
  - des opérateurs (optionnels),
  - des équations orientées (définissant les opérateurs donc optionnelles).
- soit un enrichissement :
  - des opérateurs,
  - des équations orientées (définissant les opérateurs).

avec une déclaration de variables lorsqu'elles sont nécessaires dans les équations.

Nous montrerons dans la section 5 page 109 comment dériver automatiquement un type concret (Ada) à partir de la déclaration des constructeurs, et section 3.2 page 60 comment dériver une spécification impérative (Ada) à partir de la signature et du lien d'implémentation, ainsi qu'un texte impératif à partir d'un terme algébrique comprenant des variables (paramètres formels). Ce texte impératif calcule la valeur du terme quelles que soient les valeurs de certaines de ses variables, dites d'entrée, et affecte éventuellement une représentation dans le domaine concret de la valeur algébrique du terme dans certaines de ses variables, dites de sortie, et qui peuvent être les mêmes que celles d'entrée.

La première étape à la dérivation est la transformation de toutes les équations orientées qui forment l'axiomatisation de l'opérateur *op* que nous dérivons, en une équation unique dont le terme gauche est sous forme $op(x_1, \ldots, x_n)$ et dont la partie droite effectue à la fois le filtrage des paramètres effectifs (pattern matching) et le calcul pour chaque équation d'origine, de son terme en partie droite conditionné par la réussite du filtrage en partie gauche.

Exemple 3.5: Soient les équations qui définissent l'opérateur *last* :

1 $last(cons(x, nil)) \Longrightarrow t_1[x]$
2 $last(cons(x, cons(y, l))) \Longrightarrow t_2[x, y, l]$

avec $t_1[x] = cons(x, nil)$ et $t_2[x, y, l] = last(cons(y, l))$.
Elles deviennent l'unique équation :

$last(x) ==>$ if $is\text{-}cons(x)$ then
if $is\text{-}nil(select_2^{cons}(x))$ then $t_1(select_1^{cons}(x))$
else
$t_2(select_1^{cons}(x), select_1^{cons}(select_2^{cons}(x)), select_2^{cons}(select_2^{cons}(x)))$
endif
endif

qui est égale à :
$last(x) ==>$ if $is\text{-}cons(x)$ then
if $is\text{-}nil(select_2^{cons}(x))$ then $cons(select_1^{cons}(x), nil)$
else
$last(cons(select_1^{cons}(select_2^{cons}(x)), select_2^{cons}(select_2^{cons}(x))))$
endif
endif

Il est bien sûr possible d'optimiser l'application des testeurs et sélecteurs en les factorisant et en stockant les résultats dans des variables intermédiaires lorsque plusieurs sous-structures communes d'un même paramètre formel (c'est-à-dire d'une variable en partie gauche de l'équation produite) sont testées ou sélectionnées.
■

La compilation du filtrage nécessite la connaissance des implémentations des testeurs et sélecteurs correspondants à tous les constructeurs qui apparaissent en partie gauche des équations. Ce peuvent être tous les constructeurs des sortes du domaine de l'opérateur d'intérêt, ainsi que les constructeurs des sortes des domaines de ces premiers constructeurs, et ainsi de suite récursivement.

Nous imposons donc de fournir les implémentations des testeurs et sélecteurs pour tout paquetage du domaine concret ajouté à la bibliothèque. La dérivation automatique des implémentations des constructeurs telle qu'elle est définie dans la section 5 page 109 génère également les testeurs et sélecteurs correspondants.

Cette transformation d'une spécification vers une spécification considérée comme un programme fonctionnel est décrite par une opération d'"introduction de conditionnelles closes" dans [Pep91]. Dans [Kah86], Stephan Kahrs pose le problème pour des équations non orientées où la signature est composée d'un ensemble de sortes S, d'opérateurs dont le domaine de chacun est une sorte de S, et de constantes qui sont des opérateurs à domaine vide sans notion de constructeurs. Avec des règles de transformation, guidées par des propriétés les opérateurs (commutativité, injection, ...) il permet d'obtenir (sans terminaison assurée, comme l'algorithme de Knuth-Bendix, auquel sa méthode est comparée) un système confluent de règles de réécriture, sans fixer d'ordre sur les termes.

Philippe Schnoebelen, dans [Sch88] décrit une méthode de compilation de la sélection de l'équation (pattern matching) qui vérifie au fur et à mesure la couverture du

domaine et permet d'indiquer si une partie du domaine de l'opérateur n'est pas couverte par les équations qui le définissent, ou l'est par plusieurs équations. Cette méthode a été implémentée par Philippe Turlier [Tur92] pour la compilation de Lotos vers C.

Nancy Zambrano dans [Zam95] décrit en détail ce passage vers une unique équation, sous forme d'une post-condition sur les variables paramètres formels de la signature concrète (Ada). Une pré-condition sur ces mêmes variables définit le domaine d'application de l'opérateur si celui-ci est partiel. Pré-condition et post-condition qui sont calculables à partir de la spécification d'origine, et signature concrète à fournir pour la dérivation forment une seule spécification algébrico-opérationnelle :

Exemple 3.6: En reprenant l'exemple précédent dans le cadre des spécifications algébrico-opérationnelle et avec une dérivation vers une fonction :

```
spec-op : LIST_OP
   sort of interest : LIST
   sorts : ELEM
   ref-spec : LIST
   operation :
      function LAST(x: LIST) return LIST;
      pre : is-nil(x) is-false
      post : result =
         if is-cons(x) then
              if is-nil(select_2^cons(x)) then cons(select_1^cons(x), nil)
              else
                   last(cons(select_1^cons(select_2^cons(x)), select_2^cons(select_2^cons(x))))
              endif
         endif
```

et pour une dérivation vers une procédure (qui ne permet pas d'avoir un paramètre formel en sortie uniquement) :

```
spec-op : LIST_OP
   sort of interest : LIST
   sorts : ELEM
   ref-spec : LIST
   operation :
      procedure LAST(mod x: LIST);
      pre : is-nil(x) is-false
      post : x' =
         if is-cons(x) then
             if is-nil(select_2^cons(x)) then cons(select_1^cons(x), nil)
             else
                  last(cons(select_1^cons(select_2^cons(x)), select_2^cons(select_2^cons(x))))
             endif
         endif
```

■

Nous effectuons également une transformation au niveau du source pour tous les sous-termes t du terme à dériver :

$$t \equiv f(\ldots, x, \ldots, x, \ldots)$$
par
$$t \equiv let\ v = x\ in\ f(\ldots, x, \ldots, v, \ldots)$$

avec v une variable "fraîche" de la sorte de x.

3.1.5 Formalisme

Forme du code impératif généré :

Un texte impératif dérivé d'une équation algébrique définissant un opérateur c est l'expression syntaxique du calcul d'une représentation d'une valeur algébrique (ou de plusieurs dans le cas d'un co-domaine produit cartésien). Cette valeur peut être une constante, une variable, ou en général une expression fonctionnelle (que nous définirons précisément plus loin), mais une partie du calcul de cette valeur peut être sous forme procédurale : nous pouvons devoir dériver un terme en un texte du style fonctionnel pour diverses raisons (devoir faire un `return`, par exemple), mais le texte Ada dérivé peut contenir des portions dont la seule forme possible est procédurale si l'un des sous-termes à dériver contient un constructeur ou opérateur défini implémenté par une procédure ou contenant une affectation. De plus, la partie procédurale de la dérivation de ce sous-terme devra être générée et exécutée avant le code réalisant le terme le contenant.

Nous notons :

$[\![\texttt{T}]\!]$ un texte `T` consistant en un calcul procédural : une séquence d'instructions Ada d'affectation ou d'appel de procédure.

$\langle\!|\texttt{T}|\!\rangle$ un texte `T` consistant en une expression fonctionnelle, ou en un produit cartésien d'expressions fonctionnelles : une variable, un appel de fonction, un aggrégat article ou tableau, une déréférence, une allocation, un composant indexé ou sélecté, ou une composition de ces expressions.

(P, F) le résultat de la dérivation vers un texte impératif d'un nœud du graphe obtenu par optimisation de l'arbre abstrait représentant l'axiomatisation d'un opérateur défini. Le premier élément du doublet, P, est la partie procédurale et le second, F, est la partie fonctionnelle.

Soient les fonctions *Proc* et *Fonc* telles que :

$$Proc([\![\texttt{P}]\!], \langle\!|\texttt{F}|\!\rangle) = \texttt{P}$$
$$Fonc([\![\texttt{P}]\!], \langle\!|\texttt{F}|\!\rangle) = \texttt{F}$$

Un texte dérivé ne contient jamais de partie fonctionnelle vide. L'évaluation de celle-ci, après exécution de la partie impérative éventuelle, est une représentation de la valeur du terme algébrique codé au nœud du graphe traduit.
Exemple : $([\![\texttt{read(x);}]\!], \langle\!|\texttt{x+1}|\!\rangle)$ pourrait être un résultat de dérivation.

Interprétation algébrique :

Par construction de la dérivation, à toute valeur des variables `x` d'un programme Ada, nous pouvons associer une valeur abstraite t, soit par relation entre types et sortes prédéfinis, soit par le mécanisme de construction standard des termes du chapitre 5. Cette interprétation est appelée $\mathcal{I}$. Dans le cas de base des variables, nous avons :

$$\mathcal{I}(\texttt{x}) = t$$

Pour toute expression Ada `F(x)` obtenue par dérivation, où `F` est l'image de l'opérateur abstrait f, nous avons inductivement :

$$\mathcal{I}(\texttt{F(x)}) = f(\mathcal{I}(\texttt{x}))$$

Enfin, un programme `P` modifie les variables concrètes `x` et calcule, s'il termine, un certain état de ces variables. Nous notons par :

$$\texttt{P} \Rightarrow \mathcal{I}(\texttt{F})$$

l'interprétation algébrique de `F` dans l'état produit par l'exécution de `P`.

Les combinateurs :

Nous utilisons les combinateurs tels ceux définis par [Bac78], avec les notations suivantes :

- Le combinateur π_i permet d'obtenir le $i^{ème}$ élément d'une séquence, ou le projeté du $i^{ème}$ élément d'un produit cartésien :

$$\forall i \in [1 \ldots n],\ \forall (x_1, \ldots, x_n) \cdot \pi_i(x_1, \ldots, x_n) = x_i$$

- Le combinateur ϕ est une constante : $\forall x \cdot \phi(x) = \phi$.
- Les séquences de combinateurs sont notées entre "<" et ">".
- La forme de construction est la simple application d'une séquence de combinateurs à un objet : $< f_1, \ldots, f_n >(x) = (f_1(x), \ldots, f_n(x))$.

Exemple : $< \pi_3, \phi, \pi_1 >(x, y, z, t) = (z, \phi, x)$

Définition formelle de la fonction $\Delta_D(c) = (\mathcal{E}, In, Out)$:

Soit $c : (s_1, \ldots, s_n) \rightarrow (s_{n+1}, \ldots, s_{n+m})$ un constructeur ou un opérateur défini. Dans le cas d'un constructeur, nous avons $m = 1$. Le lien Δ est identique pour un constructeur et un opérateur défini. Nous l'explicitons dans le cas d'un opérateur :

Définition 3.2: $\mathcal{E}(c) = [x_1 \ldots x_p]\{\mathrm{T}\}$ est un texte Ada dont les p variables $x_1 \ldots x_p$ sont dites liées. Nous avons $p \geq n$, $p \geq m$ et $p \leq n + m$.

$$\mathcal{E}(c) = [x_1 \ldots x_p]\{\mathrm{T}\} \Rightarrow BoundVars(\mathcal{E}(c)) = (x_1, \ldots, x_p)$$

Les variables de T autres que $x_1 \ldots x_p$ sont dites libres.

■

Définition 3.3: $In(c)$ et **$Out(c)$** sont chacun une construction de p combinateurs (p étant le nombre de variables liées de $\mathcal{E}(c)$), qui sont soit un combinateur π, soit le combinateur ϕ :

$$\begin{array}{c} In(c) = < P_1, \ldots, P_p > \ \wedge \ P_{i \in [1 \ldots p]} \in \{\pi_1, \ldots, \pi_n\} \cup \{\phi\} \\ Out(c) = < P'_1, \ldots, P'_p > \ \wedge \ P'_{i \in [1 \ldots p]} \in \{\pi_1, \ldots, \pi_m\} \cup \{\phi\} \end{array}$$

avec comme caractéristiques :

1. La représentation de chaque opérande de c est introduite dans une et une seule variable liée de $\mathcal{E}(c)$:

$$\forall\, In(c) = < P_1 \ldots P_p >,\ \forall i \in [1 \ldots n],\ \exists! j \in [1 \ldots p] \cdot P_j = \pi_i$$

2. La représentation du résultat (ou de chaque élément du résultat) est récupéré dans une seule variable liée de $\mathcal{E}(c)$:

$$\forall\, Out(c) = < P_1 \ldots P_p >,\ \forall i \in [1 \ldots m],\ \exists! j \in [1 \ldots p] \cdot P_j = \pi_i$$

3. Une variable liée de $\mathcal{E}(c)$, soit reçoit une valeur comme paramètre effectif, c'est-à-dire par $In(c)$, soit a sa valeur calculée dans $\mathcal{E}(c)$ et est considérée comme un résultat récupéré par $Out(c)$, soit elle correspond aux 2 cas précédemment cités, mais elle possède toujours une valeur après l'exécution de $\mathcal{E}(c)$:

$$\left.\begin{array}{c} \forall\, i \in [1 \ldots p] \\ \forall\, In(c) = < P_1, \ldots, P_p > \\ \forall\, Out(c) = < P'_1, \ldots, P'_p > \end{array}\right\} \cdot P_i = \phi \Rightarrow P'_i \neq \phi$$

4. Une variable liée de $\mathcal{E}(c)$ à la fois en entrée et en sortie correspond à des sortes du domaine et du co-domaine identiques :

$$\left.\begin{array}{c} \forall\, i \in [1 \ldots p] \\ \forall\, In(c) = < P_1, \ldots, P_p > \\ \forall\, Out(c) = < P'_1, \ldots, P'_p > \end{array}\right\} \cdot \begin{array}{l} P_i \neq \phi \ \wedge \ P'_i \neq \phi \Rightarrow \\ P_i(s_1 \ldots s_n) = P'_i(s_{n+1} \ldots s_{n+m}) \end{array}$$

■

$In(c)$ est le combinateur d'affectation des paramètres d'entrée (passage des paramètres effectifs) et $Out(c)$ est le combinateur d'affectation des paramètres de sortie (récupération du (ou des) résultat(s)).

Combinateur inverse des combinateurs In et Out :

Soit Pos une fonction qui renvoie l'indice de la position de son premier argument dans la séquence que constitue le deuxième argument.

Définition 3.4: Le combinateur inverse $\boldsymbol{In(c)^{-1}}$ du combinateur $In(c)$ d'un constructeur ou opérateur défini $c : (s_1, \ldots, s_n) \text{ -> } (s_{n+1}, \ldots, s_{n+m})$ avec $m = 1$ dans le cas d'un constructeur est, pour $In(c) = < P_1, \ldots, P_p >$:

$$In(c)^{-1} = < P'_1, \ldots, P'_n > \text{ tel que } P'_{i \in [1 \ldots n]} = \pi_{Pos(\pi_i, In(c))}$$

Le combinateur inverse $\boldsymbol{Out(c)^{-1}}$ du combinateur $Out(c) = < P_1, \ldots, P_p >$ d'un même constructeur ou opérateur défini est calculé de la même manière que celui du combinateur $In(c)$ (en remplaçant n par m dans la règle précédente).

Les caractéristiques 1. et 2. de la définition 3.3 qui précède nous assurent que $\forall i \in [1 \ldots n] \cdot Pos(\pi_i, In(c))$ et $\forall i \in [1 \ldots m] \cdot Pos(\pi_i, Out(c))$ ont bien une valeur.

■

Composition de combinateurs In ou Out :

Définition 3.5: Soient $< P_1, \ldots, P_p >$ et $< P'_1, \ldots, P'_q >$ deux combinateur In ou Out tels que $\forall\, \pi_j \in < P_1, \ldots, P_p > \cdot (j \in [1 \ldots q])$. Le combinateur " $\circ$ " de composition de ces deux combinateurs est défini par :

$$< P_1, \ldots, P_p > \ \circ \ < P'_1, \ldots, P'_q > = < P''_1, \ldots, P''_p > \text{ tel que :}$$
$$\forall i \in [1 \ldots p] \cdot \left(P''_i = P_i(P'_1, \ldots, P'_q) \right)$$

■

Combinateurs de superposition :

Quelle que soit la forme de dérivation (fonctionnelle ou procédurale) de l'opérateur défini c dont le texte du lien d'implémentation est $[x_1 \dots x_n]\{\mathrm{T}\}$, les variables auxquelles il faut affecter les représentations des opérandes de c sont : $In(c)^{-1}(x_1 \dots x_n)$.

Dans le cas d'une forme fonctionnelle de dérivation, en affectant à $In(c)^{-1}(x_1 \dots x_n)$ les valeurs respectives représentant $t_1 \dots t_n$, $\mathcal{E}(c) \circ In(c)(x_1 \dots x_n)$ a pour résultat une représentation de $c(t_1 \dots t_n)$. Dans le cas d'une forme procédurale de dérivation, les variables dans lesquelles il faut récupérer le résultat sont $Out(c)^{-1}(x_1 \dots x_n)$. Certaines variables pouvant être communes aux entrées et aux sorties, nous nous dotons de combinateurs permettant de faire cette composition : $\oplus_L$ et $\oplus_R$ décrivent une "superposition" de deux produits cartésiens d'expressions Ada ayant les mêmes types pour leurs éléments de même position.

Définition 3.6: Soient $(e_1 \dots e_p)$ et $(e'_1 \dots e'_p)$ des produits cartésiens dont chaque élément est, soit une expression Ada, soit ϕ, et tels que si $e_{i\in[1\dots p]} \neq \phi \ \wedge \ e'_i \neq \phi$ alors e_i et e'_i sont de même type. Les combinateurs $\oplus_L$ et $\oplus_R$ sont définis par :

$$(e_1 \dots e_p) \oplus_L (e'_1 \dots e'_p) = (e''_1 \dots e''_p) \text{ tel que : } \forall i \in [1 \dots p] \cdot \left(\begin{array}{l} e_i = \phi \ \Rightarrow e''_i = e'_i \\ e_i \neq \phi \ \Rightarrow e''_i = e_i \end{array} \right)$$

et

$$(e_1 \dots e_p) \oplus_R (e'_1 \dots e'_p) = (e''_1 \dots e''_p) \text{ tel que : } \forall i \in [1 \dots p] \cdot \left(\begin{array}{l} e'_i = \phi \ \Rightarrow e''_i = e_i \\ e'_i \neq \phi \ \Rightarrow e''_i = e'_i \end{array} \right)$$

■

Nous utilisons les combinateurs $\oplus_L$ et $\oplus_R$ pour décrire la composition du passage de paramètres effectifs par In avec la récupération des résultats dans ces paramètres par Out.

Combinateurs de soustraction :

Soit un combinateur qui permet de supprimer d'un produit cartésien les éléments qui appartiennent à un autre produit cartésien :

Définition 3.7: Soient $(e_1 \dots e_p)$ et $(e'_1 \dots e'_p)$ des produits cartésiens dont chaque élément est, soit une expression Ada, soit ϕ, et tels que aucun produit cartésien n'a plusieurs occurrences d'une même expression. Le combinateur $\ominus$ est défini par :

$$(e_1 \dots e_p) \ominus (e'_1 \dots e'_p) = (e''_1 \dots e''_p) \text{ tel que :}$$
$$\forall i \in [1 \dots p] \cdot \left(\begin{array}{l} e_i \in \{e'_1 \dots e'_p\} \ \Rightarrow e''_i = \phi \\ e_i \notin \{e'_1 \dots e'_p\} \ \Rightarrow e''_i = e_i \end{array} \right)$$

■

Combinateurs de choix :

Lors de la superpositions de 2 produits cartésiens, nous pouvons nous retrouver avec plusieurs occurrences du même élément dans le produit cartésien résultat, même si chaque produit cartésien initial n'a pas plusieurs occurrences d'une même variable. Ceci peut être un inconvénient lorsqu'il s'agit de déterminer un produit cartésien de variables qui doivent être la destination d'un résultat.

Nous nous dotons alors de deux combinateurs qui permettent de calculer une superposition de 2 produits cartésiens D_1 et D_2 sans occurrences multiples qui soit un produit cartésien également sans occurrences multiples, et privilégiant un opérande ou l'autre :

$$D_1 \boxplus_L D_2 = D_1 \oplus_L (D_2 \ominus D_1)$$
$$D_1 \boxplus_R D_2 = (D_1 \ominus D_2) \oplus_R D_2$$

Paramètres effectifs modifiés ou non :

Soit $t = c(t_1, \ldots, t_n)$ ou $t = c(t')$ un terme à dériver dont le constructeur ou opérateur défini c devient une procédure. Soit la dérivation de ses opérandes (ou de son unique opérande de type produit cartésien) : $\Delta^{\#}(t_1) = ([\![p_1]\!], \langle\!| f_1 |\!\rangle), \ldots, \Delta^{\#}(t_n) = ([\![p_1]\!], \langle\!| f_n |\!\rangle)$ ou $\Delta^{\#}(t') = ([\![p']\!], \langle\!| f_1, \ldots, f_n |\!\rangle)$. Les paramètres effectifs de $\mathcal{E}(c)$ doivent prendre pour valeurs, les valeurs respectives de $f_1, \ldots, f_n$. Chaque paramètre peut être l'expression correspondante de $f_1, \ldots, f_n$ (généralement une variable) mais lorsqu'il n'est pas une expression-gauche (un paramètre formel en mode `in`, ou un appel de fonction, par exemple), une recopie dans une variable intermédiaire doit être faite.

Les paramètres qui sont modifiés par l'implémentation de c sont déterminés par :

$$P^c_{In\text{-}Out}(f_1, \ldots, f_n) = \{f_{i \in [1 \ldots n]} \mid f_i \in Out(c)^{-1} \circ In(c)(f_1, \ldots, f_n)\}$$

Les paramètres non modifiés par l'implémentation de c sont déterminés par :

$$P^c_{In}(f_1, \ldots, f_n) = \{f_{i \in [1 \ldots n]} \mid f_i \notin Out(c)^{-1} \circ In(c)(f_1, \ldots, f_n)\}$$

Nous notons *In-to-Out* le combinateur qui permet d'obtenir les paramètres d'entrée correspondants en sortie aux résultats $(t_{n+1}, \ldots, t_{n+m})$ de l'évaluation de t, ou ϕ sinon :

$$In\text{-}to\text{-}Out(c) = Out(c)^{-1} \circ In(c)$$

Nous notons *Out-to-In* le combinateur qui permet d'obtenir, pour une destination fixée du résultat (plusieurs expression-gauches devant contenir chaque élément de la représentation du produit cartésien résultat), leur correspondance en tant que paramètre effectif d'entrée (modifié) :

$$Out\text{-}to\text{-}In(c) = In(c)^{-1} \circ Out(c)$$

Exemple 3.7: Soit l'opérateur *dblappend* dont la spécification est la suivante :

```
operators
  dblappend : (seqnat, seqnat, seqnat) -> (seqnat, seqnat)

equations
  dblappend(L1, L2, L3) ==> ( append(L1,L2), append(L1,L3) )
```

Soit le lien de dérivation de *dblappend* avec une procédure `dblconcat` qui renvoie la représentation du doublet résultat séparément, dans le premier paramètre qui est ainsi modifié, et dans un paramètre supplémentaire en quatrième position :

$\mathcal{E}(dblappend) =$
$\quad [lmod1, ladd1, ladd2, lres2]\{$`dblconcat(lmod1, ladd1, ladd2, lres2)`$\}$
$In(dblappend) = < \pi_1, \pi_2, \pi_3, \phi >$
$Out(dblappend) = < \pi_1, \phi, \phi, \pi_2 >$

L'interface de la procédure qui implémente *dblappend* est la suivante :

```
procedure dblconcat(lmod1: in out listint;
                    ladd1, ladd2: in listint;
                    lres2: out listint) ;
```

Nous avons :

$$\begin{aligned} \textit{In-to-Out}(dblappend) &= Out(dblappend)^{-1} \circ In(dblappend) \\ &= < \pi_1, \phi, \phi, \pi_2 >^{-1} \circ < \pi_1, \pi_2, \pi_3, \phi > \\ &= < \pi_1, \pi_4 > \circ < \pi_1, \pi_2, \pi_3, \phi > \\ &= < \pi_1, \phi > \end{aligned}$$

$$\begin{aligned} \textit{Out-to-In}(dblappend) &= In(dblappend)^{-1} \circ Out(dblappend) \\ &= < \pi_1, \pi_2, \pi_3, \phi >^{-1} \circ < \pi_1, \phi, \phi, \pi_2 > \\ &= < \pi_1, \pi_2, \pi_3 > \circ < \pi_1, \phi, \phi, \pi_2 > \\ &= < \pi_1, \phi, \phi > \end{aligned}$$

■

3.2 Principe de la dérivation

3.2.1 De la signature algébrique à l'interface impérative

L'idée intuitive pour la dérivation de l'axiomatisation d'un opérateur défini est de simuler le calcul abstrait par un calcul concret.

Nous avons pour l'opérateur $op : (s_1 \dots s_n) \text{ -> } (s_{n+1} \dots s_{n+m})$ l'unique équation orientée suivante où $\alpha_1 \dots \alpha_n$ sont des variables de types respectifs $s_1 \dots s_n$:

$$op(\alpha_1 \dots \alpha_n) \Longrightarrow \mathrm{T}_{[\alpha'_1 \dots \alpha'_n]}$$
$$\text{avec } \{\alpha'_1 \dots \alpha'_n\} \subseteq \{\alpha_1 \dots \alpha_n\}$$

Exemple :

Pour $op : (elem, seq[elem], seq[elem]) \text{ -> } seq[elem]$ l'équation
$op(x, y, z) \Longrightarrow append(ins(x, y), z)$

Pour plus de clarté, nous avons supposé que toutes les variables $\alpha_1 \dots \alpha_n$ apparaissent dans T et nous notons $\mathrm{T}_{[\alpha_1 \dots \alpha_n]}$. L'absence de certaines variables de $\alpha_1 \dots \alpha_n$ dans T ne changeant rien à la dérivation. Nous avons également le lien décrivant le profil de dérivation de op défini par $\mathcal{E}(op)$, $In(op)$ et $Out(op)$

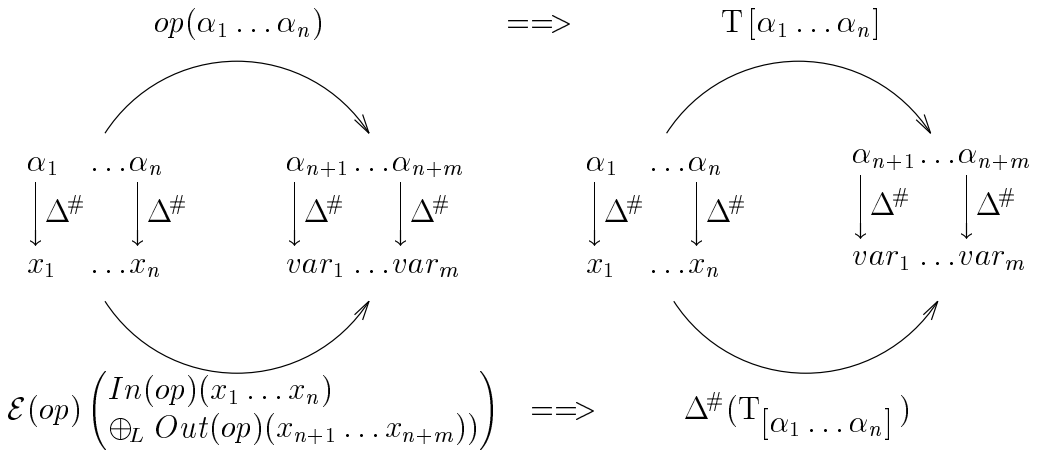
Nous proposons la méthode suivante pour la fonction $\Delta^{\#}$ de dérivation du code Ada à partir de l'équation vue précédemment.

$$\boxed{\Delta^{\#}(op(\alpha_1 \dots \alpha_n)) \Longrightarrow \Delta^{\#}(\mathrm{T}_{[\alpha_1 \dots \alpha_n]})}$$

Nous posons $\Delta^{\#}((\alpha_1 \dots \alpha_n)) = (x_1 \dots x_n)$ et $op(\alpha_1 \dots \alpha_n) = (\alpha_{n+1} \dots \alpha_{n+m})$.
Soient $(x_{n+1} \dots x_{n+m})$ de nouvelles variables de types respectifs $\Delta_S(s_{n+1}) \dots \Delta_S(s_{n+m})$.

Pour un opérateur dont le lien spécification LPG – interface Ada est défini, la règle de calcul de la figure 3.5 détermine les assertions concernant les variables en paramètres effectifs de la fonction ou procédure dérivée. En appliquant cette règle, nous pouvons déterminer l'en-tête du programme Ada, ainsi que les variables des paramètres formels qui doivent représenter le résultat algébrique en dérivation impérative. La figure 3.3 suivante montre le passage du calcul abstrait au calcul concret :

FIG. 3.3 – *Du calcul abstrait au calcul concret*



avec $(var_1 \dots var_m) \equiv Out(op)^{-1}(In(op)(x_1 \dots x_n) \oplus_L Out(op)(x_{n+1} \dots x_{n+m}))$

À partir de l'équation définissant un opérateur, et de son lien d'implémentation, nous pouvons déterminer sa spécification impérative, c'est-à-dire son profil (fonction ou procédure, mode des paramètres formels) et dans quels paramètres formels doivent se trouver en fin d'exécution du texte dérivé, une représentation du résultat algébrique calculé par la partie droite de l'équation.

De manière à garder une bonne lisibilité du texte impératif dérivé, nous conservons comme noms des paramètres formels de la fonction ou procédure dérivée, les noms des paramètres en partie gauche de l'équation définissant l'opérateur à dériver : $\Delta^{\#}(\alpha_1, \dots, \alpha_n) = (\alpha_1, \dots, \alpha_n)$ dans la figure 3.3, où $(\alpha_1, \dots, \alpha_n)$ sont alors des variables.

La spécification concrète (Ada) de la procédure ou fonction se calcule de la manière qui suit :

Le style : une procédure si $\exists P \neq \phi \ \cdot \ (Out(op) = < \dots, P, \dots >)$ et une fonction sinon.

Les types des paramètres : $(par\text{-}type_1, \dots, par\text{-}type_p)$ tels que :

$$(par\text{-}type_1, \dots, par\text{-}type_p) = In(op)(\Delta_S(s_1), \dots, \Delta_S(s_n)) \oplus_L Out(op)(\Delta_S(s_{n+1}), \dots, \Delta_S(s_{n+m}))$$

Les modes des paramètres : $(par\text{-}mode_1, \dots, par\text{-}mode_p)$ tels que :

$$\forall i \in [1 \dots p] \cdot \left(\begin{array}{l} In(op)_i \neq \phi \ \wedge \ Out(op)_i = \phi \Rightarrow par\text{-}mode_i = \texttt{in} \\ In(op)_i \neq \phi \ \wedge \ Out(op)_i \neq \phi \Rightarrow par\text{-}mode_i = \texttt{in out} \\ In(op)_i = \phi \ \wedge \ Out(op)_i \neq \phi \Rightarrow par\text{-}mode_i = \texttt{out} \end{array} \right)$$

Les noms des paramètres : Pour une meilleure lisibilité, nous conservons les noms des paramètres formels de *op* comme paramètres formels de la fonction ou pro-

cédure dérivée `OP` (sinon, le même renommage devrait être appliqué à la partie droite de l'équation avant dérivation) :

$$\begin{aligned}(&par\text{-}nom_1 \ldots par\text{-}nom_n)\\ &= In(op)(\Delta^{\#}(\alpha_1), \ldots, \Delta^{\#}(\alpha_n)) \oplus_L Out(op)(x_{n+1}, \ldots, x_{n+m})\\ &= In(op)(\alpha_1, \ldots, \alpha_n) \oplus_L Out(op)(x_{n+1}, \ldots, x_{n+m})\end{aligned}$$

Et la spécification dans le domaine concret obtenue est :

- pour une procédure :

$$\texttt{procedure}\ \mathcal{E}(op)(par\text{-}nom_1 : par\text{-}mode_1\ par\text{-}type_1;\ \ldots\ ;\ par\text{-}nom_p : par\text{-}mode_p\ par\text{-}type_p\ )$$

- pour une fonction, où $p = n$:

$$\texttt{function}\ \mathcal{E}(op)(par\text{-}nom_1 : par\text{-}type_1 \ldots par\text{-}nom_p : par\text{-}type_p)\texttt{return}\ \Delta_S(s_{n+1})$$

Les variables $\alpha_1, \ldots, \alpha_n$ sont pour certaines des paramètres formels en entrée uniquement : $P^{op}_{In}(\alpha_1, \ldots, \alpha_n)$ et pour d'autres des paramètres formels en entrée-sortie : $P^{op}_{In\text{-}Out}(\alpha_1, \ldots, \alpha_n)$. L'ensemble des premières est appelé $Param_{In}$ dans la dérivation de $\mathrm{T}_{[\alpha_1, \ldots, \alpha_n]}$ et l'ensemble des secondes : $Param_{In\text{-}Out}$. La dérivation de $\mathrm{T}_{[\alpha_1, \ldots, \alpha_n]}$ se fait avec, dans le cas d'une procédure, connaissance de la destination des résultats : $(var_1 \ldots var_m)$. La dérivation du texte impératif en tient compte, et ajoute les affectations des résultats aux paramètres formels $(var_1 \ldots var_m)$ dans le code dérivé (en une seconde étape, nous pouvons attribuer à chaque sous-terme de T sa destination lorsque celle-ci est calculable à partir de la destination $(var_1 \ldots var_m)$ de T et des liens d'implémentation des opérateurs de chaque sous-terme). Nous notons $\Delta^{\#}{}_{(var_1 \ldots var_m)}$ cette dérivation.

Exemple 3.8: Soit l'opérateur

$$op : (seqnat, seqnat, seqnat, seqnat) \text{ -> } (seqnat, seqnat, seqnat)$$

défini par l'équation :

$$op(x, y, z, t) \Longrightarrow (concat(concat3(x, y, renverse(z)), t),\ concat(x, z),\ tri(concat(t, x)))$$

dont nous voulons faire la dérivation vers une procédure `OP`.

En chapitre 7 section 7.2 nous proposons une syntaxe pour la définition du lien d'implémentation. Le lien d'implémentation (et de dérivation de op) est le suivant :

$$\begin{aligned}&\mathcal{E}(op) = [x, y, z, t]\{\texttt{OP(x,y,z,t)}\},\\ &In(op) = <\pi_1, \pi_2, \pi_3, \pi_4>, Out(op) = <\pi_2, \phi, \pi_1, \pi_3>\end{aligned}$$

Soit $\Delta_S(seqnat) =$ `listint`, les types $(par\text{-}type_1, \ldots, par\text{-}type_4)$ des paramètres respectifs de la procédure `OP` sont :

$$\begin{aligned}&In(op)(\Delta_S(seqnat), \Delta_S(seqnat), \Delta_S(seqnat), \Delta_S(seqnat))\\ &\oplus_L Out(op)(\Delta_S(seqnat), \Delta_S(seqnat), \Delta_S(seqnat))\\ &= (\texttt{listint}, \texttt{listint}, \texttt{listint}, \texttt{listint})\end{aligned}$$

Leurs modes respectifs ($par\text{-}mode_1, \ldots, par\text{-}mode_4$) sont (`in out`, `in`, `in out`, `in out`).

Les paramètres formels de `OP` sont :

$$\begin{aligned}(par\text{-}nom_1 \ldots par\text{-}nom_4) &= In(op)(x,y,z,t) \oplus_L Out(op)(v_1,v_2,v_3,v_4)\\ &= (x,y,z,t)\end{aligned}$$

les variables $v_{[1\ldots4]}$ ne servent pas car il n'y a pas de paramètre en mode `out` uniquement.

La partie gauche de l'équation est dérivée dans le domaine concret en :

`procedure OP(`$par\text{-}nom_1$ `:` $par\text{-}mode_1$ $par\text{-}type_1 \ldots par\text{-}nom_4$ `:` $par\text{-}mode_4$ $par\text{-}type_4$`)`
`= procedure`
`OP(x : in out listint; y : in listint; z : in out listint; t : in out listint)`

et le terme T en partie droite de l'équation à traduire, doit être dérivé en un texte Ada qui, en fin d'exécution, doit contenir les représentations des résultats algébriques dans les variables suivantes (ou être une expression fonctionnelle si la destination calculée est ϕ) :

$$\begin{aligned}Out(op)^{-1}(par\text{-}nom_1,\ldots,par\text{-}nom_4) &= <\pi_2,\phi,\pi_1,\pi_3>^{-1}(par\text{-}nom_1,\ldots,par\text{-}nom_4)\\ &= <\pi_3,\pi_1,\pi_4>(x,y,z,t)\\ &= (z,x,t)\end{aligned}$$

En conséquence, le texte dérivé est noté $\Delta^{\#}{}_{(z,x,t)}\,(\mathrm{T})$.

■

3.2.2 Mise en œuvre du lien Δ dans la dérivation par $\Delta^{\#}$

Soient :

- $c : (s_1,\ldots,s_n) \text{ -> } (s_{n+1},\ldots,s_{n+m})$ un opérateur défini.
- $\Delta_D(c) = (\mathcal{E}(c), In(c), Out(c))$ le lien de dérivation ou d'implémentation de c.
- un terme algébrique de LPG ayant c en racine :
  $$t = c(t_1,\ldots,t_n) \text{ avec } t_1 : s_1,\ \ldots,\ t_n : s_n$$
  ou
  $$t = c(t') \text{ avec } t' = d(u_1,\ldots,u_k) : (s_1,\ldots,s_n)$$
- $(\widehat{t_1},\ldots,\widehat{t_n})$ le produit cartésien résultat de l'évaluation de $(t_1,\ldots,t_n)$ ou de t' dans un modèle choisi de la sémantique adoptée pour nos spécifications algébriques. Le résultat de t dans le modèle choisi est $\widehat{t}$.
- les types qui correspondent aux sortes du domaine et du co-domaine de c :
  $$typ_1 = \Delta_S(s_1),\ \ldots\ ,\ typ_n = \Delta_S(s_n)$$
  et
  $$typ_{n+1} = \Delta_S(s_{n+1}),\ \ldots\ ,\ typ_{n+m} = \Delta_S(s_{n+m})$$
- $x_1 \ldots x_n$ des variables de types respectifs $typ_1 \ldots typ_n$.

La correction du lien d'implémentation implique :

Dans le cas d'une dérivation vers une forme fonctionnelle, c'est-à-dire pour $Out(c) = <\phi, \ldots, \phi>$, $m = 1$ et les variables $x_1 \ldots x_n$ ayant pour valeurs respectives des représentations des opérandes de c, les valeurs algébriques $\widehat{t_1} \ldots \widehat{t_n}$:

$$\Delta^{\#}(t) = \mathcal{E}(c) \circ In(c)(x_1, \ldots, x_n)$$

et la valeur de l'évaluation de $\Delta^{\#}(t)$ est une représentation de $\widehat{t}$.
D'après l'interprétation définie en 3.1.5 nous pouvons écrire :

$$\{\texttt{v :=}\ \Delta^{\#}(t);\ \} \Rightarrow \mathcal{I}(v) = \widehat{t}$$

Ce qui donne, en intégrant la dérivation des opérandes de c :

- pour $t = c(t_1, \ldots, t_n)$ avec $\Delta^{\#}(t_1) = (\llbracket p_1 \rrbracket, \langle\!| f_1 |\!\rangle), \ldots, \Delta^{\#}(t_n) = (\llbracket p_n \rrbracket, \langle\!| f_n |\!\rangle)$ et $\forall i \in [1 \ldots n] \cdot\ p_i \Rightarrow \mathcal{I}(f_i) = \widehat{t_i}$

  $$\Delta^{\#}(t) = (\llbracket p_1; \ldots; p_n \rrbracket, \langle\!| \mathcal{E}(c) \circ In(c)(f_1, \ldots, f_n) |\!\rangle)$$

  et la valeur de l'évaluation de $\mathcal{E}(c) \circ In(c)(f_1, \ldots, f_n)$ après l'exécution de $p_1; \ldots; p_n;$ est égale à une représentation de $\widehat{t}$:

  $$\{p_1; \ldots; p_n;\} \Rightarrow \mathcal{I}(\mathcal{E}(c) \circ In(c)(f_1, \ldots, f_n)) = \widehat{t}$$

- pour $t = c(t')$ avec $\Delta^{\#}(t') = (\llbracket p';\rrbracket, \langle\!| f_1, \ldots, f_n |\!\rangle)$ et $p' \Rightarrow \mathcal{I}(f') = \widehat{t'}$

  $$\Delta^{\#}(t) = (\llbracket p';\rrbracket, \langle\!| \mathcal{E}(c) \circ In(c)(f_1, \ldots, f_n) |\!\rangle)$$

  et la valeur de l'évaluation de $\mathcal{E}(c) \circ In(c)(f_1, \ldots, f_n)$ après l'exécution de p'; est égale à une représentation de $\widehat{t}$:

  $$\{p';\} \Rightarrow \mathcal{I}(\mathcal{E}(c) \circ In(c)(f_1, \ldots, f_n)) = \widehat{t}$$

Dans le cas d'une dérivation vers une forme procédurale, c'est-à-dire lorsque $\exists P \neq \phi \cdot (Out(c) = < \ldots, P, \ldots >)$ et $m \geq 1$: soient $x_{n+1} \ldots x_{n+m}$ des variables de types respectifs $typ_{n+1} \ldots typ_{n+m}$, par abus de notation, nous pouvons écrire :

$$\Delta^{\#}(t) = \{\mathcal{E}(c)(In(c)(x_1, \ldots, x_n) \oplus_L Out(c)(x_{n+1}, \ldots, x_{n+m}));\}$$

et après exécution de $\Delta^{\#}(t)$, chaque élément du produit cartésien $(\widehat{t_{n+1}}, \ldots, \widehat{t_{n+m}})$ est représenté respectivement par les variables :

$$Out(c)^{-1}(In(c)(x_1, \ldots, x_n) \oplus_L Out(c)(x_{n+1}, \ldots, x_{n+m}))$$

En intégrant la dérivation des opérandes de c, tels qu'ils sont décrits dans la dérivation fonctionnelle (plusieurs opérandes, ou un unique opérandes dont la sorte est un produit cartésien de sortes simples) :

- pour $t = c(t_1, \ldots, t_n)$ avec $\Delta^{\#}(t_1) = (\llbracket p_1 \rrbracket, \langle\!| f_1 |\!\rangle), \ldots, \Delta^{\#}(t_n) = (\llbracket p_n \rrbracket, \langle\!| f_n |\!\rangle)$:

  $$\Delta^{\#}(t) = (p^t, f^t)$$

avec

$$p^t = [\![\ p_1; \ldots; p_n; \\ \bigcup_{i\in[1\ldots n]}\{x_i := f_i;\ |\ f_i \in P_{In\text{-}Out}(f_1,\ldots,f_n)\ \wedge\ \neg\, LeftExpr(f_i)\}; \\ \mathcal{E}(c)(In(c)(f'_1,\ldots,f'_n) \oplus_L Out(c)(x_{n+1},\ldots,x_{n+m}));]\!]$$

$$f^t = \langle\!|Out(c)^{-1}(In(c)(f'_1,\ldots,f'_n) \oplus_L Out(c)(x_{n+1},\ldots,x_{n+m}))|\!\rangle$$

tels que

$$\forall i \in [1\ldots n]\cdot\quad (f_i \in P_{In\text{-}Out}(f_1,\ldots,f_n)\ \wedge\ \neg\, LeftExpr(f_i)) \Rightarrow f'_i = x_i \\ \wedge\ \neg(f_i \in P_{In\text{-}Out}(f_1,\ldots,f_n)\ \wedge\ \neg\, LeftExpr(f_i)) \Rightarrow f'_i = f_i$$

et après exécution de p^t, les variables de f^t contiennent une représentation de $\widehat{t}$:

$$\{p^t;\} \Rightarrow \mathcal{I}(f^t) = \widehat{t}$$

- pour $t = c(t')$ avec $\Delta^{\#}(t') = ([\![p';]\!], \langle\!|f_1,\ldots,f_n|\!\rangle)$:

$$\Delta^{\#}(t) = (p^t, f^t)$$

avec

$$p^t = [\![\ p'; \\ \bigcup_{i\in[1\ldots n]}\{x_i := f_i;\ |\ f_i \in P_{In\text{-}Out}(f_1,\ldots,f_n)\ \wedge\ \neg\, LeftExpr(f_i)\}; \\ \mathcal{E}(c)(In(c)(f'_1,\ldots,f'_n) \oplus_L Out(c)(x_{n+1},\ldots,x_{n+m}));]\!]$$

$$f^t = \langle\!|Out(c)^{-1}(In(c)(f'_1,\ldots,f'_n) \oplus_L Out(c)(x_{n+1},\ldots,x_{n+m}))|\!\rangle$$

tels que

$$\forall i \in [1\ldots n]\cdot\quad (f_i \in P_{In\text{-}Out}(f_1,\ldots,f_n)\ \wedge\ \neg\, LeftExpr(f_i)) \Rightarrow f'_i = x_i \\ \wedge\ \neg(f_i \in P_{In\text{-}Out}(f_1,\ldots,f_n)\ \wedge\ \neg\, LeftExpr(f_i)) \Rightarrow f'_i = f_i$$

et après exécution de p^t, les variables de f^t contiennent une représentation de $\widehat{t}$:

$$\{p^t;\} \Rightarrow \mathcal{I}(f^t) = \widehat{t}$$

Dérivation avec détermination des paramètres effectifs en utilisant la connaissance de la destination du résultat :

Les règles précédentes, décrites dans le cas d'une dérivation procédurale ne nécessitent que :

- le lien de dérivation $\Delta(c)$ du constructeur ou de l'opérateur défini $c : (s_1,\ldots,s_n) \text{ -> } (s_{n+1},\ldots,s_{n+m})$,

- le résultat $\Delta^{\#}(t_1)\ldots\Delta^{\#}(t_n)$ de la dérivation des opérandes $t_1\ldots t_n$ de c,

- des variables "fraîches" $(x_1,\ldots,x_n)$, c'est-à-dire n'appartenant ni à $\mathcal{E}(c)$, ni à $\Delta^{\#}(t_1),\ldots,\Delta^{\#}(t_n)$, de types respectifs $\Delta_S(s_1),\ldots,\Delta_S(s_n)$. Elles serviront de variables intermédiaires en position de paramètres effectifs modifiés (`in out`) à la place d'expressions de $\Delta^{\#}(t_1)\ldots\Delta^{\#}(t_n)$ qui ne sont pas des expression-gauches,

- des variables "fraîches" $(x_{n+1},\ldots,x_{n+m})$, c'est-à-dire n'appartenant ni à $\mathcal{E}(c)$, ni à $\Delta^{\#}(t_1),\ldots,\Delta^{\#}(t_n)$, de types respectifs $\Delta_S(s_{n+1}),\ldots,\Delta_S(s_{n+m})$. Ces variables ne seront utilisées comme paramètres effectifs que pour les paramètres formels de $\mathcal{E}(c)$ qui sont en sortie uniquement : $Out(c)^{-1}(BoundVars(\mathcal{E}(c)))$.

Dans le cas où les destinations des éléments du résultat sont connus (des expression-gauches), il nous est alors possible de déterminer leur position en tant que paramètres effectifs de $\mathcal{E}(c)$, et lorsque ceux-ci sont à la fois en entrée et en sortie, de choisir entre :

Cas n^o 1 l'affectation à la destination de l'expression résultat de la compilation de l'opérande, et d'utiliser la destination comme paramètre effectif.

Cas n^o 2 l'utilisation de l'expression résultat de la compilation de l'opérande de c comme paramètre effectif de c (lorsqu'il s'agit d'une expression-gauche) et de l'affectation de celle-ci, modifiée, à la destination après exécution de $\mathcal{E}(c)$.

Soient les expressions-gauches $dest_1,\ldots,dest_m$ (généralement des variables) où $dest_i = \phi$ si la destination n'est pas connue, qui doivent contenir chacune après exécution de $\Delta^{\#}(c(t_1,\ldots,t_n))$, une représentation d'un élément du produit cartésien résultat. Si $dest_{i\in[1\ldots m]} = \phi$ alors l'expression-gauche contenant un élément du résultat doit être déduite des paramètres d'entrée. Soit $(d_1,\ldots,d_n) \equiv \mathit{Out\text{-}to\text{-}In}(c)(dest_1,\ldots,dest_m)$ le produit cartésien dont chaque élément est, soit ϕ, soit l'expression-gauche qui doit contenir une représentation du résultat en fin d'exécution de $\Delta^{\#}(c(t_1,\ldots,t_n))$, et dans sa position de paramètre effectif modifié.

Cas n^o 1, le paramètre effectif est déterminé selon la règle suivante :

$$\begin{aligned}
\forall i \in [1\ldots n]\ \cdot\quad & \begin{pmatrix} f_i \in P_{\mathit{In\text{-}Out}}(f_1,\ldots,f_n) \\ \wedge\ d_i \neq \phi \end{pmatrix} \Rightarrow \text{paramètre effectif } d_i \\
& \wedge \begin{pmatrix} f_i \in P_{\mathit{In\text{-}Out}}(f_1,\ldots,f_n) \\ \wedge\ d_i = \phi \\ \wedge\ \neg\, \mathit{LeftExpr}(f_i) \end{pmatrix} \Rightarrow \text{paramètre effectif } x_i \\
& \wedge \begin{pmatrix} f_i \notin P_{\mathit{In\text{-}Out}}(f_1,\ldots,f_n) \\ \vee\ \begin{pmatrix} f_i \in P_{\mathit{In\text{-}Out}}(f_1,\ldots,f_n) \\ \wedge\ d_i = \phi \\ \wedge\ \mathit{LeftExpr}(f_i) \end{pmatrix} \end{pmatrix} \Rightarrow \text{paramètre effectif } f_i
\end{aligned}$$

Cas n^o 2, le paramètre effectif est déterminé selon la règle suivante :

$$\forall i \in [1 \ldots n] \cdot \quad \begin{pmatrix} f_i \in P_{In\text{-}Out}(f_1, \ldots, f_n) \\ \wedge\ \neg\, LeftExpr(f_i) \\ \wedge\ d_i \neq \phi \end{pmatrix} \Rightarrow \text{paramètre effectif } d_i$$
$$\wedge \begin{pmatrix} f_i \in P_{In\text{-}Out}(f_1, \ldots, f_n) \\ \wedge\ \neg\, LeftExpr(f_i) \\ \wedge\ d_i = \phi \end{pmatrix} \Rightarrow \text{paramètre effectif } x_i$$
$$\wedge \begin{pmatrix} f_i \notin P_{In\text{-}Out}(f_1, \ldots, f_n) \\ \vee\ LeftExpr(f_i) \end{pmatrix} \Rightarrow \text{paramètre effectif } f_i$$

Nous montrons maintenant les règles qui définissent dans ces deux cas, le code dérivé pour exécuter dans le domaine concret l'application de l'opérateur ou constructeur c.

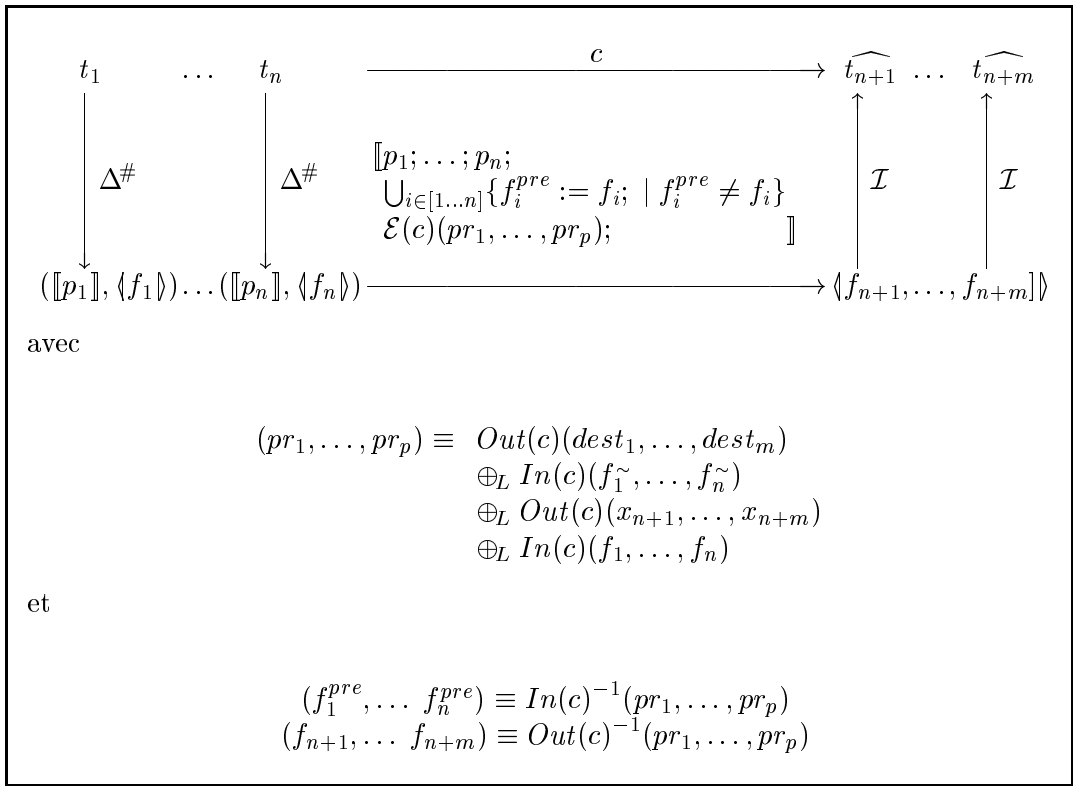
- Soient $pr_1, \ldots, pr_p$ les paramètres effectifs assignés aux variables liées de $\mathcal{E}(c)$.

- Soient $f_1^{\sim}, \ldots, f_n^{\sim}$ les parties fonctionnelles de $\Delta^{\#}(t_1), \ldots, \Delta^{\#}(t_n)$ qui sont des expression-gauches, et peuvent donc être en position de paramètre effectif modifié :

$$\forall i \in [1 \ldots n] \cdot \quad \begin{array}{l} LeftExpr(f_i) \Rightarrow f_i^{\sim} = f_i \\ \wedge\ \neg\, LeftExpr(f_i) \Rightarrow f_i^{\sim} = \phi \end{array}$$

- Soient $(f_1^{pre}, \ldots, f_n^{pre})$ les paramètres d'entrée déterminés en fonction des paramètres effectifs de $\mathcal{E}(c)$.

La règle de dérivation 3.4 suivante montre le **cas n^o 1** où les destinations connues sont utilisées en priorité par rapport aux expression-gauches $f_{i \in [1 \ldots n]}$ dérivées, pour être modifiées par la partie procédurale de $\Delta^{\#}(c(t_1, \ldots, t_n))$.



avec

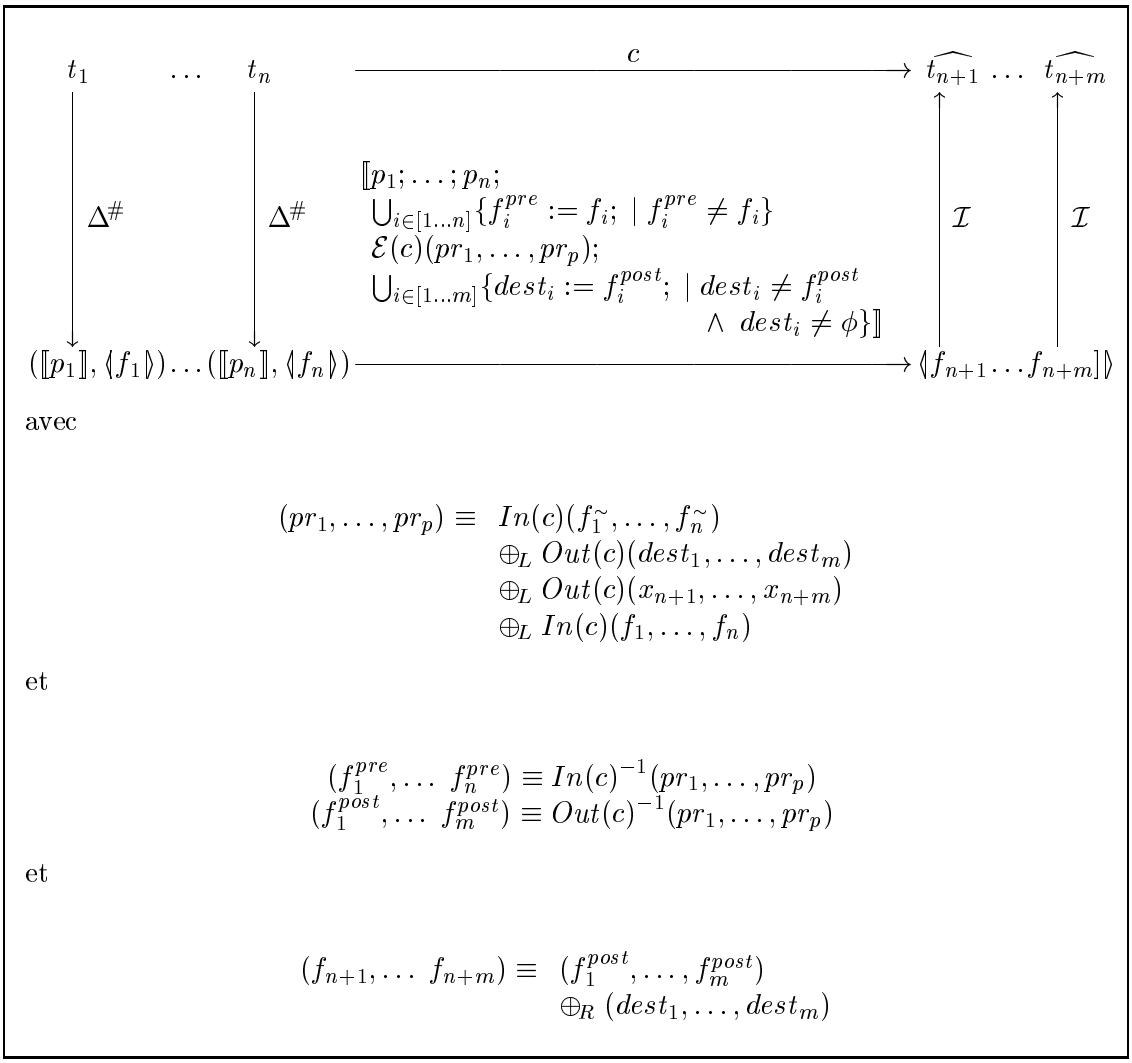
$$\begin{aligned}(pr_1, \ldots, pr_p) \equiv\ & Out(c)(dest_1, \ldots, dest_m) \\ & \oplus_L In(c)(f_1^{\sim}, \ldots, f_n^{\sim}) \\ & \oplus_L Out(c)(x_{n+1}, \ldots, x_{n+m}) \\ & \oplus_L In(c)(f_1, \ldots, f_n)\end{aligned}$$

et

$$\begin{aligned}(f_1^{pre}, \ldots\ f_n^{pre}) &\equiv In(c)^{-1}(pr_1, \ldots, pr_p) \\ (f_{n+1}, \ldots\ f_{n+m}) &\equiv Out(c)^{-1}(pr_1, \ldots, pr_p)\end{aligned}$$

FIG. 3.4 – *Traduction dans le domaine concret de l'application d'un opérateur, destinations connues des résultats en paramètres effectifs.*

La règle de dérivation 3.5 suivante montre le **cas n°2** où les expression-gauches $f_{i\in[1\ldots n]}$ dérivées sont utilisées en priorité aux destinations connues pour être modifiées par la partie procédurale de $\Delta^\#(c(t_1, \ldots, t_n))$. Soient $(f_1^{post}, \ldots, f_n^{post})$ les paramètres de sortie déterminés en fonction des paramètres effectifs de $\mathcal{E}(c)$.



avec

$$(pr_1, \ldots, pr_p) \equiv \begin{array}{l} In(c)(f_1^{\sim}, \ldots, f_n^{\sim}) \\ \oplus_L \, Out(c)(dest_1, \ldots, dest_m) \\ \oplus_L \, Out(c)(x_{n+1}, \ldots, x_{n+m}) \\ \oplus_L \, In(c)(f_1, \ldots, f_n) \end{array}$$

et

$$\begin{array}{l} (f_1^{pre}, \ldots \ f_n^{pre}) \equiv In(c)^{-1}(pr_1, \ldots, pr_p) \\ (f_1^{post}, \ldots \ f_m^{post}) \equiv Out(c)^{-1}(pr_1, \ldots, pr_p) \end{array}$$

et

$$(f_{n+1}, \ldots \ f_{n+m}) \equiv \begin{array}{l} (f_1^{post}, \ldots, f_m^{post}) \\ \oplus_R \, (dest_1, \ldots, dest_m) \end{array}$$

FIG. 3.5 – *Traduction dans le domaine concret de l'application d'un opérateur, expression-gauches dérivées des opérandes en paramètres effectifs*

Conclusion

Nous avons présenté dans ce chapitre :

- le cadre pratique dans lequel se place la technique de dérivation ;
- une description semi-formelle du lien d'implémentation d'une spécification algébrique par un paquetage impératif;
- une énumération des différentes contraintes à respecter lors de la dérivation d'un terme algébrique ;

- une description du pré-traitement qui consiste à compiler la sélection d'équation ("pattern matching") pour obtenir une unique équation avec le terme à dériver en partie droite ;
- le formalisme utilisé pour décrire le code impératif obtenu par la dérivation d'un terme algébrique ;
- le formalisme qui permet de définir les différents composants d'un lien d'implémentation entre une spécification algébrique et un paquetage impératif ;
- le principe global de la dérivation et de l'obtention de l'interface du paquetage impératif à partir du profil algébrique et de l'équation à dériver ;
- les différents cas d'un pas de dérivation, suivant les différents styles d'implémentation d'un opérateur, les différents types de résultats de la dérivation de ses opérandes, et selon que les adresses destinations des résultats intermédiaires soient synthétisées ou héritées.

Chapitre 4

Résolution des conflits d'accès aux variables, avec un coût minimum en duplications

4.1 Motivations, notations, principe

Soit l'axiomatisation de l'opérateur *op* définie par l'équation :

$$op(x_1 \dots x_n) ==> \mathrm{T}_{(x_1 \dots x_n)}$$

La dérivation de $op(x_1 \dots x_n)$ avec l'information du profil de *op*, tel que cela est montré en fin du chapitre précédent, permet d'obtenir l'interface de la fonction ou procédure qui implémente *op*.

Pour obtenir le corps de programme correspondant à l'interface, nous devons dériver le texte Ada qui permet d'évaluer le terme $\mathrm{T}_{(x_1 \dots x_n)}$ dans le domaine concret. Pour simplifier, nous conserverons les mêmes noms de variables pour les paramètres formels : $(x_1, \dots, x_n) = \Delta^{\#}(x_1, \dots, x_n)$.

4.1.1 Variable = valeur référencée par un nom

Dans le domaine des spécifications algébriques, nous ne manipulons que des valeurs. Une variable x du terme T est donc :

- soit une des variables $x_{i \in [1 \dots n]}$ liées universellement,
- soit une variable intermédiaire définie par un "*let* $x = t_1$ *in* t_2 *endlet*", sous-terme de T.

Dans le premier cas, la valeur de x est celle que x reçoit en tant que paramètre formel en partie gauche de l'équation, et qui est constante pendant tout le calcul du terme en partie droite. Dans le second cas, x reçoit la valeur du terme t_1, et possède cette valeur pendant tout le calcul du terme t_2.

Conflits d'accès aux variables

Les sous-termes de T qui consistent en une application d'un opérateur à des variables peuvent avoir comme code Ada équivalent, des instructions qui détruisent les valeurs de ces variables :

Exemple 4.1: Soit l'opérateur d'intérêt

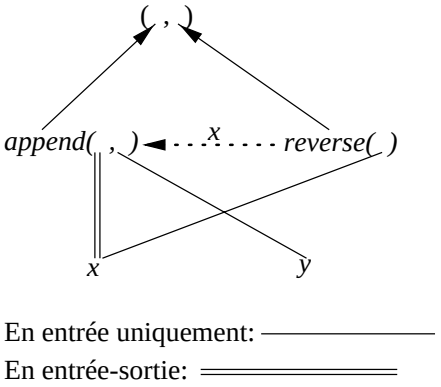
$$apprev : (seqint, seqint) \rightarrow (seqint, seqint)$$

qui renvoie un produit cartésien dont le premier élément est une concaténantion des deux séquences d'entiers opérandes, et le second élément, la séquence d'entiers en premier opérande rangés dans l'ordre inverse :

$$apprev(x, y) \Longrightarrow (append(x, y), reverse(x))$$

Soient les implémentations de *append* par la procédure `APPEND` qui modifie son premier paramètre, et de l'opérateur *reverse* par la fonction `REVERSE`.

Le terme à dériver est représenté par le graphe suivant :



Les utilisations des variables sont indiquées par les traits simples ou doubles, les dépendances fonctionnelles sont indiquées par les flèches simples. La signification des flèches en pointillés est donnée plus loin.

Nous voyons simplement que l'exécution de l'implémentation du *append* avant celle du *reverse* modifie la variable x, et que sa valeur n'est plus correcte lors de l'appel de la fonction `REVERSE` (exception faite lorsque y est la liste vide).

Des textes Ada possibles sont les suivants :

Incorrect	Correct	Correct

`APPEND(x,y) ;`
`v1 := REVERSE(x) ;`
	`v1 := REVERSE(x) ;`
	`APPEND(x,y) ;`
		`DUPLICATION(x,x2) ;`
		`APPEND(x,y) ;`
		`v1 := REVERSE(x2) ;`

	Résultats dans : `x`, `v1`	Résultats dans : `x`, `v1`

■

La résolution d'un conflit d'accès à une variable v s'effectue :

1. soit, pour une instruction i qui modifie v,
   (a) en utilisant une unique duplication de v pour toutes les instructions qui l'utilisent après i en lecture seulement,
   (b) puis en utilisant à la place de v, une duplication dans une variable intermédiaire pour chaque instruction j qui l'utilisait et modifiait v après i,
2. soit en plaçant chaque instruction qui utilise v en lecture seulement avant toutes les instructions qui la modifient.

Un ensemble de conflits sur une ou plusieurs variables peut se résoudre en utilisant l'une des deux méthodes pour certains accès conflictuels, et l'autre méthode pour les accès conflictuels restants. Il est clair que la seconde méthode est la plus économique, et c'est cet ordre de précédence des calculs que nous voulons privilégier et qui est indiqué dans l'exemple 4.1 précédent par un arc orienté en pointillés (la variable conflictuelle est indiquée sur l'arc). Le but de notre travail est de résoudre ces conflits en respectant au maximum l'ordre indiqué par ces arcs $\cdots\overset{x}{\cdots}\blacktriangleright$ pour effectuer les calculs dans le domaine concret.

Nous effectuons l'optimisation qui consiste à utiliser une donnée comme paramètre effectif avant que sa valeur soit modifiée. Nous ne descendons pas à un niveau de détail élevé qui consisterait à étudier quelles sont les parties exactes d'une donnée qui sont utilisées et lesquelles sont modifiées. Ceci peut être exprimé au niveau des algèbres en désignant des sous-termes au moyen de sélecteurs appliqués aux termes. Cette approche plus fine n'est pas traitée dans ce mémoire et nous nous limitons à une granularité du niveau des variables, qui sont dites utilisées ou modifiées totalement ou non.

Dans le domaine abstrait, au niveau algébrique, seul l'*ordre de dépendance fonctionnel* doit obligatoirement être respecté dans l'évaluation d'un terme : chaque sous-terme doit être évalué avant l'opération racine du terme qui le contient.

Nous considérons que l'ordre d'évaluation entre opérandes est une convention du langage. Lorsque cette évaluation dépend du résultat d'un autre opérande, nous l'appelons *ordre de dépendance opérationnel*. C'est le cas du *if-then-else* et du *let* qui sont traités à part des autres opérateurs. Les opérandes sont supposés ne pas pouvoir faire des "*effets de bord*".

De manière à appliquer cette première optimisation, nous considérons également que la propriété d'ordre quelconque d'évaluation des opérandes est vérifiée par le code généré. Il est alors possible de réarranger les séquences de code entre elles lorsqu'elles proviennent de sous-termes disjoints, de manière à éviter les duplications de sauvegarde des variables. Cette technique peut être comparée à celles d'*anticipation* et de *temporisation* de [MR74] pour optimiser le temps d'exécution et l'espace mémoire utilisé.

4.1.2 Localisation des conflits

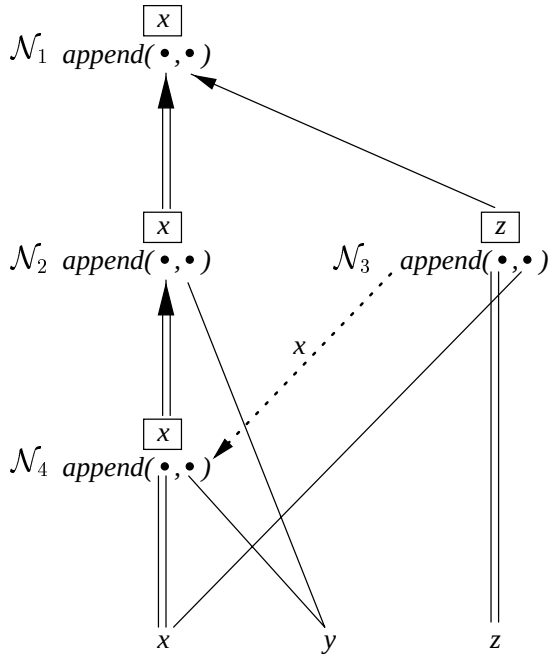
Le mode : utilisation ou une modification, d'une variables x qui apparaît dans le terme T à dériver : $t = op(\ldots, x, \ldots)$ et $t \in \mathrm{T}$, peut être déduite du profil de la fonction

ou procédure qui implémente *op*. Par contre cette variable peut également être utilisée ou modifiée par un calcul lorsqu'elle contient le résultat du calcul d'un opérande (en entrée-sortie ou sortie uniquement dans le texte Ada qui calcule cet opérande) : $t' = op'(\ldots, t, \ldots)$ et $t' \in \mathrm{T}$ avec x en entrée-sortie dans l'implémentation de *op*.

Nous proposons dans une première étape de ne pas introduire en sortie uniquement dans un texte Ada dérivé, de variable des paramètres formels $x_1, \ldots, x_n$, et de ne reconnaître les utilisations et modifications des ces variables qu'aux positions de T où elles apparaissent explicitement comme opérandes :

Exemple 4.2: Soit le terme $\mathrm{T}_{[x,\,y,\,z]}$ à dériver qui suit, pour lequel l'opérateur *append* possède la même implémentation que précédemment :

$$\mathrm{T}_{[x,\,y,\,z]} = append(append(append(x, y), y), append(z, x))$$



Pour chaque calcul, la variable qui en contient le résultat est encadrée.

Voici trois solutions possibles du conflit d'accès à x :

`APPEND(z,x);`	`DUPLICATION(x,x2);`	`DUPLICATION(x,x2);`
`APPEND(x,y);`	`APPEND(x,y);`	`APPEND(x2,y);`
`APPEND(x,y);`	`APPEND(x,y);`	`APPEND(x2,y);`
`APPEND(x,z);`	`APPEND(z,x2);`	`APPEND(z,x);`
	`APPEND(x,z);`	`APPEND(x2,z);`
Résultat dans : `x`	Résultat dans : `x`	Résultats dans : `x2`

■

Lorsque nous résolvons un conflit d'accès, dans le domaine concret, à une variable qui est un opérande dans T (dans le domaine des spécifications), cette variable n'est

pas source de conflits aux niveaux "supérieurs" (dans les calculs postérieurs), même lorsqu'elle apparaît dans le texte généré, contenant le résultat d'un sous-calcul, et à nouveau modifiée. Dans notre exemple, la résolution du conflit d'accès à x, entre les calculs codés aux nœuds $\mathcal{N}_4$ et $\mathcal{N}_3$ résolvent les conflits qui pourraient exister entre les calculs des nœuds $\mathcal{N}_2$ et $\mathcal{N}_3$, ou $\mathcal{N}_1$ et $\mathcal{N}_3$. Nous ne déduisons donc pas d'ordre de précédence entre ces calculs.

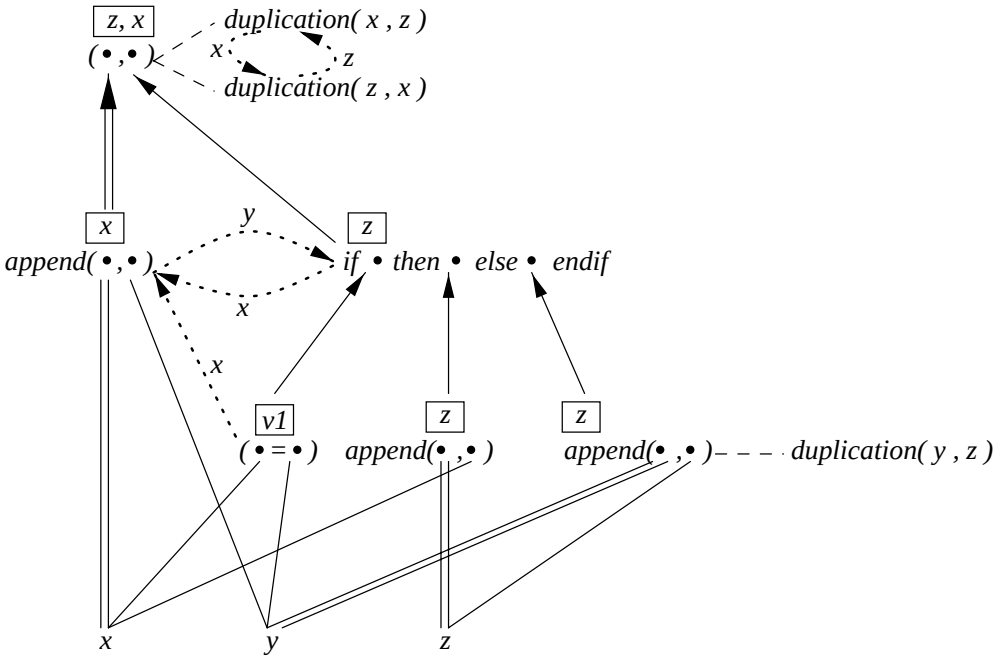
Nous pouvons donc introduire des précédences entre calculs déduits des utilisations/modifications, au niveau concret, des opérandes qui sont des variables au niveau abstrait. Ceci est vrai tant que les paramètres effectifs du code dérivé de $op(t_1, \ldots, t_m)$ sont "synthétisés" à partir des résultats du code dérivé de $t_1, \ldots, t_m$ (parmi ces paramètres, les seules variables de $\{x_1, \ldots, x_n\}$ sont celles qui contiennent un résultat de $t_1, \ldots, t_m$, et sont à la position de paramètre qui leur correspond. Elles ne sont pas introduites sous forme de paramètre effectif en sortie uniquement).

4.1.3 Sous-ensembles d'accès conflictuels indépendants

Il est possible de partitionner les calculs en sous-calculs qui n'entrent pas en conflit d'accès aux variables, malgré leur utilisation des mêmes variables selon des modes différents :

Exemple 4.3: Le terme T doit ici être dérivé en un corps de procédure qui calcule deux valeurs. Les paramètres formels en sortie de la procédure qui contiennent ces valeurs sont z et x, dans cet ordre. L'opérateur *append* possède la même implémentation que dans l'exemple précédent :

$$\mathrm{T}_{[x,\,y,\,z]} = (append(x, y), \text{if } (x = y) \text{ then } append(z, x) \text{ else } append(y, z) \text{ endif })$$



Pour chaque calcul, la variable qui en contient le résultat est encadrée.

Le résultat du calcul du *if-then-else* doit être contenu dans une unique variable destination, car cette variable doit être utilisée comme paramètre du calcul dont le *if-then-else* est opérande. Nous rajoutons donc en fin des calculs *then* et/ou *else* des affectations qui impliquent cette destination unique du résultat (ici : choix de z et affectation de y à z dans le sous-terme *else*).

De même, nous rajoutons également des affectations au calcul effectué en racine du terme à dériver, lorsque le code à obtenir est celui d'une procédure et que paramètres formels de retour sont alors connus (ici : les résultats des calculs contenus dans x et z doivent être respectivement dans z et x).

Nous appelons ces affectations des *post-affectations*.

■

Les calculs des sous-termes *then* et *else* ne sont jamais effectués tous les deux. Conséquences :

- les sous-termes *then* et *else* n'ont pas de conflits entre eux,
- ne sachant pas quel sous-terme, *then* ou *else* est exécuté, nous ne déduisons pas de conflits entre ceux-ci et les sous-termes "externes" au *if-then-else*. Nous réglons alors *localement* les conflits internes aux sous-termes *then* et *else*. Nous attribuons ensuite l'ensemble de leurs utilisations et modifications de variables au terme *if-then-else* (conflits d'accès à x et y dans notre exemple).

Les post-affectations peuvent être également sujettes à conflits, comme c'est le cas dans notre exemple. Un seul ordre de précédence doit ici être supprimé, et la variable conflictuelle dupliquée (par exemple : `v2 := x; x := z; z := v2`). Nous voyons qu'il s'agit d'un autre type de conflit, à résoudre *localement* aux post-affectations. Il ne porte pas sur la valeur des variables à l'appel de la procédure, mais sur la valeur d'un résultat de sous-calcul : il n'y a pas lieu dans notre exemple de déduire un conflit entre les post-affectations de la "racine" et les sous-termes qui utilisent et modifient x et z.

4.1.4 Raisons et conséquences du choix d'un ordre d'évaluation entre plusieurs sous-termes

Soient 2 sous-termes t_1 et t_2 d'un terme T. Chacun est dérivé vers un texte impératif qui peut comporter des affectations destructives de variables ou des appels de procédures. Ce que nous appelons "ordre d'évaluation" d'un terme par rapport à un autre est l'ordre d'exécution du texte dérivé de l'un par rapport à celui dérivé de l'autre. Cet ordre peut avoir plusieurs motivations, qui sont répertoriées selon les cas suivants :

1. L'ordre d'évaluation de t_1 par rapport à t_2 est fixé par la dépendance des calculs : si t_1 est sous-terme de t_2, le texte Ada dérivé de t_1 doit être évalué avant celui dérivé de t_2. Ce qui revient à dire que le texte (instruction prédéfinie, d'affectation, appel de fonction ou de procédure) qui implémente l'opérateur en racine du terme t_1 doit être exécutée avant celui qui implémente l'opérateur en racine du terme t_2.

2. Lorsque 2 sous-termes t_1 et t_2 n'ont ni dépendance fonctionnelle ni dépendance opérationnelle entre eux, il peut être intéressant de leur affecter cependant un ordre d'exécution du code dérivé en fonction des conflits d'utilisation de leurs variables communes. En effet chacune peut induire un ordre suivant son mode d'utilisation dans les texte Ada dérivés de t_1 et t_2, par la nécessité d'effectuer des duplications de sauvegarde ou non pour résoudre les conflits d'utilisation. Notre but est de minimiser ces recopies de sauvegarde :

   (a) Si la variable est uniquement *utilisée* par tous les textes dérivés : l'ordre est quelconque.

   (b) Si la variable est *modifiée* par toutes les instructions l'utilisant dans les textes Ada dérivés, une seule instruction pourra effectivement la modifier. Il faudra choisir le sous-terme précis dont l'opérateur en racine sera dérivé en une instruction modifiant cette variable, et faire une duplication de cette variable dans des variables intermédiaires que modifieront les autres instructions concernées. L'ordre d'évaluation est donc quelconque (seule possibilité d'optimisation : réutiliser les variables intermédiaires après leur modification par les autres instructions modificatrices, mais ces techniques de réutilisation d'espace mémoire sont bien connues, implantées dans les compilateurs et nous ne les reprendrons pas ici).

   (c) Si la variable est *utilisée* par certains sous-termes et *modifiée* par d'autres, la recopie sera évitée s'il n'existe qu'une instruction modificatrice dérivée, et que toutes les instructions utilisatrices peuvent être exécutées avant. Le non respect de cet ordre implique une duplication de sauvegarde de la variable. Une exception sera faite lorsque la variable provient d'un paramètre en mode `in` uniquement, puisque ne pouvant être modifiée, elle devra inévitablement être dupliquée dans une variable intermédiaire.

3. Pour résoudre un conflit d'affectation de plusieurs variables : lorsqu'un ensemble de variables doit recevoir les valeurs d'un autre ensemble de variables, il y a conflit si l'intersection des 2 ensembles n'est pas vide. L'ordre des affectations doit alors être calculé (par exemple si x, y, z doivent respectivement recevoir les valeurs de a, z, x), et des duplications de variables éventuellement générées (par exemple si x, y, z doivent respectivement recevoir les valeurs de z, x, y). Ces cas peuvent arriver lorsque l'on doit ajouter des affectations pour que le résultat de l'exécution d'un texte dérivé soit rangé dans une destination précise (cf la règle 3.5 page 69) : paramètres formels de retour de l'opérateur d'intérêt, ou pour que les résultats des textes dérivés des sous-termes "*then*" et "*else*" d'un même terme "*if-then-else*" soient contenus dans les mêmes variables.

4.1.5 Arbre abstrait

Nous codons en un arbre abstrait le terme T en partie droite de l'équation définissant l'opérateur d'intérêt. Chaque nœud représente un terme dont les sous-termes

opérandes sont les sous-arbres "fils" de ce nœud. Chaque nœud de cet arbre est indicé et est noté $\mathcal{N}_i$.

Les arcs orientés d'un nœud de l'arbre $\mathcal{N}_i$ vers un nœud $\mathcal{N}_j$ signifient que le terme codé en $\mathcal{N}_i$ doit être évalué avant $\mathcal{N}_j$. Nous considérons donc comme des arcs toutes les branches de l'arbre abstrait, orientées des nœuds opérandes $\mathcal{N}_i$ vers le nœud père $\mathcal{N}_j$. L'ordre défini ainsi est appelé ordre de dépendance fonctionnelle.

Nous ajoutons à l'arbre abstrait tous les arcs représentant les ordres de dépendance dus à la sémantique des opérateurs, tels le "*if-then-else*" ou le "*let*", dont les opérandes ne sont pas évalués dans un ordre quelconque. L'ordre défini ainsi est appelé ordre de dépendance opérationnelle :

- à nœud qui code un "*let* $x = t_1$ *in* t_2 *endlet*", nous ajoutons des arcs dont l'origine est le nœud qui code t_1 et dont les extrémités sont les nœuds qui codent des sous-termes de t_2 qui n'ont pas de sous-terme à calculer (aucun opérande, ou uniquement des variables).

- à un nœud qui code un "*if-then-else*", nous devrions ajouter les arcs dont l'origine est le nœud qui code la condition booléenne, et dont les extrémités sont les nœuds des sous-termes "*then*" et "*else*" qui n'ont que des variables en opérandes, ou aucun opérande. Nous ne le faisons pas car nous avons montré en section 4.1.3 page 75 que la résolution des conflits ne peut se faire globalement entre des sous-termes d'un "*then*" ou d'un "*else*" et des termes qui ne sont pas sous-termes du même "*then*" ou "*else*". Le nœud "*if-then-else*" est considérés comme un seul nœud dont le texte dérivé de son opérateur utilise et modifie les variables utilisées et modifiées par ses sous-arbres "*then*" et "*else*".

L'arbre abstrait est décoré comme suit :

$\mathcal{N}_i.operateur$ est l'opérateur en racine du sous-terme représenté par le nœud i.

$\mathcal{N}_i.variable$ est la variable représentée par le nœud i, qui est alors une feuille.

$\mathcal{N}_i.operandes$ est la liste des nœuds qui représentent les sous-termes de profondeur 1 du terme représenté par $\mathcal{N}_i$ (ces nœuds représentent les opérandes de $\mathcal{N}_i.operateur$).

$\mathcal{N}_i.op\text{-}dependance$ est la liste des nœuds qui doivent être évalués avant le nœud $\mathcal{N}_i$ pour des raisons dues à la sémantique opérationnelle d'un opérateur prédéfini (nœud définissant une égalité dans un "*let*" avant le nœud définissant la valeur calculée). Les arcs ainsi définis sont ne peuvent être supprimés.

$\mathcal{N}_i.preaffect$ est la liste des nœuds qui codent des affectations aux variables qui sont paramètres effectifs d'entrée du code dérivé pour $\mathcal{N}_i.operateur$. Ces affectations sont effectuées avant l'appel de ce code (cf la règle 3.5 page 69).

$\mathcal{N}_i.In$ est la liste des variables qui contiennent la valeur des opérandes de $\mathcal{N}_i.operateur$, ceci *après* exécution des pré-affectations. Ce sont les variables qui sont paramètres effectifs d'entrée du code dérivé pour $\mathcal{N}_i.operateur$.

$\mathcal{N}_i$. *Out* est la liste des variables qui contiennent le résultat après l'exécution du code dérivé pour $\mathcal{N}_i$.*operateur*, c'est-à-dire ses paramètres effectifs de sortie, *avant* les post-affectations.

$\mathcal{N}_i$. *postaffect* est la liste des nœuds qui codent des affectations des paramètres effectifs de sortie du code dérivé pour $\mathcal{N}_i$.*operateur*, à des variables dont nous voulons qu'elles contiennent ce résultat (des destinations connues). Ces affectations sont effectuées après l'appel de ce code (cf la règle 3.5 page 69).

$\mathcal{N}_i$. *destination* est la liste des variables qui contiennent une représentation de la valeur du terme algébrique codé dans l'arbre dont $\mathcal{N}_i$ est la racine (après les post-affectations).

$\mathcal{N}_i$. *antecedents* indique la liste des nœuds dont il serait avantageux d'avoir exécuté le code dérivé avant celui de $\mathcal{N}_i$: c'est une liste de doublets tel $(\vec{x}, j)$, $\vec{x}$ étant une liste de variables à sauvegarder si l'ordre d'exécution n'est pas respecté, et j l'indice du nœud dont le code dérivé doit être exécuté avant celui de $\mathcal{N}_i$.

$\mathcal{N}_i$. *sauvegardes* indique la liste des nœuds qui effectuent des duplications de sauvegarde de variables opérandes dans le terme codé au nœud $\mathcal{N}_i$ avant l'exécution de son texte dérivé.

$\mathcal{N}_i$.*anticipe* est un booléen qui indique que le nœud doit avoir son code dérivé exécuté par anticipation (c'est-à-dire que le nœud est un nœud antécédent d'un autre nœud).

À partir des informations des attributs, nous allons calculer des ordres d'évaluation entre nœuds. Chaque ordre supplémentaire, ajouté à l'arbre sous forme d'arcs orientés (dont l'origine est un nœud codé dans les "*antecedents*" de la destination) correspond à un ordre d'évaluation qui économise une ou plusieurs recopies. Lorsqu'un cycle est rencontré parmi l'ensemble des arcs du graphe ainsi créé, cela signifie que tous les ordres d'évaluation ne peuvent être respectés. Un arc au moins devra être supprimé, impliquant une ou plusieurs duplications. Nous devrons alors choisir quel est celui ou ceux des arcs qu'il est possible d'ôter du graphe pour supprimer le ou les cycle(s), générant un moindre coût en duplications.

De manière à pouvoir choisir un ordre d'évaluation optimal des sous-termes, nous ne fixerons pas d'ordre d'évaluation de l'arbre représentant T (en profondeur ou en largeur d'abord, de gauche à droite ou inversement parmi les paramètres). La seule contrainte est de respecter la dépendance fonctionnelle et opérationnelle des calculs. Une évaluation de gauche à droite et en profondeur d'abord est choisie pour l'exemple et n'interviendra pas dans l'optimisation mais dans la mise en forme du texte Ada dérivé uniquement.

4.1.6 Principe

Soit A un arbre abstrait qui code un terme T (nous employons le terme d'"arbre" car nous ne considérons que les nœuds qui codent les calculs, et non les feuilles qui représentent les variables, et que l'on peut voir partagées ou non). Soit G un graphe orienté

contenant l'arbre A, auquel nous ajoutons les arcs qui codent les ordres de précédence souhaitables entre calculs. Les ordres codés par le arcs de G sont compatibles avec l'ordre de réduction applicatif (celui des langages de programmation conventionnels, puisque nous avons choisi Ada comme langage du domaine concret).

Notations : Un produit cartésien de variables sera noté $\vec{v}$. Chaque arc du graphe G du nœud $\mathcal{N}_i$ vers le nœud $\mathcal{N}_j$ est défini par une relation $\mathcal{R}(\mathcal{N}_i, \mathcal{N}_j)$:

Définition 4.1:

$\mathcal{R}_T(\mathcal{N}_i, \mathcal{N}_j)$ est la relation entre les nœuds $\mathcal{N}_i$ et $\mathcal{N}_j$ due à la dépendance des calculs. Nous considérons à la fois les ordres de dépendance fonctionnelle (arcs de A orientés du nœud opérande $\mathcal{N}_i$ vers le père $\mathcal{N}_j$) et ceux de dépendance opérationnelle (de manière à respecter la sémantique des opérateurs prédéfinis. Dans le terme "*let* $x = t_1$ *in* t_2" par exemple, le calcul de la valeur t_1 affectée à la variable x doit précéder le calcul de tout sous-terme de t_2 dont x est opérande).

$\mathcal{R}^{\vec{v}}_{<}(\mathcal{N}_i, \mathcal{N}_j)$ est la relation entre les nœuds $\mathcal{N}_i$ et $\mathcal{N}_j$ signifiant que l'évaluation du texte engendré à partir du nœud $\mathcal{N}_i$ avant celui engendré à partir du nœud $\mathcal{N}_j$ est moins coûteuse car elle évite de dupliquer les variables $\vec{v}$. Les arcs orientés $\mathcal{N}_i \rightarrow \mathcal{N}_j$ représentent cette relation de précédence.

■

Par abus de langage, nous notons $\mathcal{R}_<$ l'ensemble des arcs définis par cette relation, et $\mathcal{R}_T$ l'ensemble de ceux définis par la dépendance des calculs (ensemble des arcs de l'arbre abstrait). Nous notons également $\mathcal{R}_* = \mathcal{R}_T \cup \mathcal{R}_<$. Nous désignons par $\mathcal{R}(\mathcal{N}_i, \mathcal{N}_j)$ un arc quelconque de $\mathcal{R}_*$ lorsque la distinction entre son appartenance à $\mathcal{R}_T$ ou à $\mathcal{R}_<$ n'a pas d'importance (ni l'éventuelle variable partagée dans le cas d'un arc de $\mathcal{R}_<$).

Les étapes suivantes de la résolution des conflits d'accès sont appliqués à G. Nous noterons $\mathcal{N}_{Init}$ le nœud du graphe G qui représente le terme codé (il s'agit de $\mathcal{N}_1$ lorsque G code le terme T en partie droite de l'équation à dériver).

L'algorithme de résolution des conflits d'accès aux variables procède en plusieurs étapes :

1. Nous calculons pour chaque nœud $\mathcal{N}_i$ les attributs *destinations*, les pré- et post-affectations et les variables utilisées et modifiées par tous les nœuds du sous-arbre dont $\mathcal{N}_i$ est la racine. Ces attributs sont synthétisés à partir des attributs des nœuds opérandes. Ils sont calculés en un parcours de tous les nœuds du graphe, en profondeur d'abord :

   - Calcul pour chaque nœud des pré- et post-affectations telles qu'elles sont décrites par la règle 3.5 page 69. Ceci permet d'ajouter à l'ensemble des calculs codés dans l'arbre abstrait, les affectations supplémentaires nécessaires aux affectations du résultat final aux paramètres formels de sortie (quand il y a lieu), ainsi que les affectations nécessaires à la dérivation d'un nœud "*if-then-else*" pour avoir une destination commune aux résultats des deux calculs des sous-termes "*then*" et "*else*".

- Calcul de l'ensemble des doublets $\mathcal{N}_i \times \mathcal{N}_j$ tels que le nœud $\mathcal{N}_i$ utilise des variables $\vec{x}$ qui sont modifiées par le nœud $\mathcal{N}_j$. Nous notons $\mathcal{R}^{\vec{x}}_{<}(\mathcal{N}_i, \mathcal{N}_j)$ la relation ainsi définie. Le respect de l'ordre d'évaluation de ces nœuds détermine une duplication ou non de la variable. Les arcs orientés $\mathcal{N}_i \rightarrow \mathcal{N}_j$ sont ajoutés à l'arbre abstrait en attribuant les nœuds $\mathcal{N}_i$ à l'attribut *antécédents* de $\mathcal{N}_j$.

Nous calculons à partir des utilisations et modifications qui sont obtenues au nœud $\mathcal{N}_{Init}$, les relations $\mathcal{R}_<$ entre nœuds qui doivent utiliser une variable avant sa modification par un autre nœud.

2. Calcul de l'ensemble des cycles qui peuvent exister dans le graphe que nous venons de construire.

3. Recherche de l'ensemble des arcs qui doivent être enlevés pour supprimer les cycles. Chaque arc enlevé signifiant une duplication de variable, les arcs sont choisis de manière à minimiser le coût en recopie (pour simplifier, le calcul de ce coût sera estimé au nombre de variables recopiées, et l'algorithme tient compte des cycles ayant des parties communes).

4. Traduction du graphe de calcul obtenu (d.a.g.) en un corps de programme à la syntaxe Ada. Nous ferons une interprétation de l'ensemble du graphe où, pour chaque nœud évalué nous générerons le texte impératif correspondant.

Le graphe orienté sans cycle $G' = \mathit{ResConf}(G)$ équivalent à G, et dont la dérivation dans le domaine concret est sans conflit d'accès aux variables est tel que :

- $\forall \mathcal{N} \in G \cdot \mathcal{N} \in G'$, et la valeur calculée au nœud $\mathcal{N}$ dans le graphe G' est celle calculée dans le graphe G,
- $\forall\, \mathcal{R}(\mathcal{N}, \mathcal{N}') \in G$ un arc orienté, $\mathcal{R}(\mathcal{N}, \mathcal{N}') \in G'$,
- $\forall \mathcal{N} \in G'$ tel que la variable x est modifiée $\mathcal{N}$, $\forall \mathcal{N}' \in G'$, nous avons :
  - soit $\mathcal{N}'$ n'utilise ni ne modifie x,
  - soit $\mathcal{N}'$ utilise ou modifie x, mais $\mathcal{N}$ et $\mathcal{N}'$ appartiennent, l'un à un arbre qui code un sous-terme "*then*", l'autre à un arbre qui code le sous-terme "*else*" d'un même "*if-then-else*",
  - soit $\mathcal{N}'$ utilise x sans la modifier, et il existe une séquence de n nœuds de G' $[\mathcal{N}_{i_1}, \ldots, \mathcal{N}_{i_n}]$ tels que :

$$\mathcal{N}_{i_1} = \mathcal{N}',\ \mathcal{N}_{i_n} = \mathcal{N},\ \text{et}\ \forall j \in [1, \ldots, n-1]\ \exists\, \mathcal{R}(\mathcal{N}_{i_j}, \mathcal{N}_{i_{j+1}}) \in G'$$

  Remarque : $\mathcal{R}(\mathcal{N}_{i_j}, \mathcal{N}_{i_{j+1}})$ est un arc de $\mathcal{R}_T \cup \mathcal{R}_<$.

Conséquences : Soit $G' = \mathit{ResConf}(G)$ un graphe orienté qui code un terme T. Chaque nœud code un sous-terme de T, et les arcs orientés codent les ordres d'exécution entre les textes engendrés de chaque nœud pour calculer la valeur du sous-terme qui y

est codé. Le graphe G' est sans conflit d'accès à une variable entre les textes engendrés de chaque nœud :

- Quelle que soit le variable x, paramètre formel de l'opérateur d'intérêt ou variable définie par un *let*,
  - si x est à la fois utilisée et modifiée dans G', alors les utilisations se font avant les modifications,
  - si x est modifiée dans G', alors elle n'est utilisée qu'une seule fois en mode entrée-sortie pour sa valeur de paramètre formel de l'opérateur d'intérêt.
- Soient t et u deux sous-termes disjoints de T, G_t et G_u leurs graphes respectifs, tels que les conflits internes à G_t doivent être résolus indépendamment. Le graphe $G'_t = ResConf(G_t)$ doit être considéré comme un unique nœud, et s'il existe un conflit d'accès à une variable x entre un nœud $\mathcal{N}_i$ de G'_t et un nœud $\mathcal{N}_j$ de G_u, il doit être résolu, soit en fixant un ordre de précédence entre les exécutions des codes engendrés pour G'_t et $\mathcal{N}_j$, soit en effectuant une duplication de sauvegarde de x. Si G'_t n'est pas choisi pour utiliser ou modifier x, une unique duplication, en un nœud $duplication(x, x_2)$ ajouté à G_u, est nécessaire. Nous remplaçons alors toute occurence de x dans G'_t par x_2.

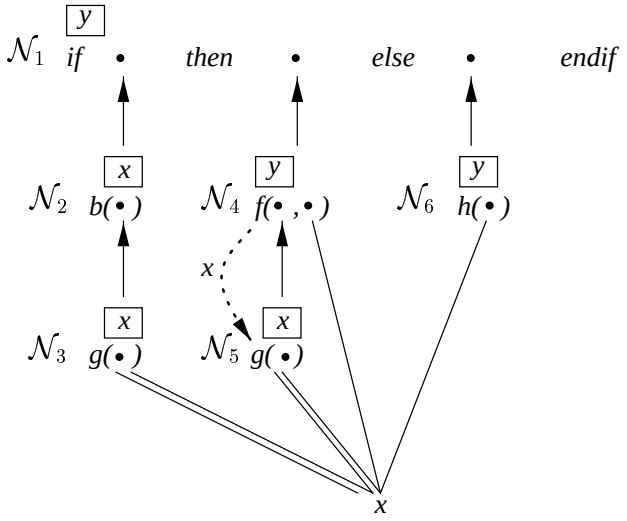
Exemple de l'application récursive de l'algorithme de résolution de conflits :

Exemple 4.4: Soit le terme à dériver :

T = if $b(g(\underline{x}))$ then
  $f(g(\underline{x}), x)$
else
  $h(x)$
endif

Les variables qui sont modifiées dans le domaine concret par le code engendré pour appliquer l'implémentation de l'opérateur ou du constructeur dont elles sont opérandes, sont soulignées. L'opérateur g, par exemple est implémenté par une procédure G qui modifie son unique paramètre.

Soit le codage de T en un arbre abstrait où les nœuds sont les suivants :



Les destinations des résultats du "*then*" et du "*else*" sont supposées calculées (de manière à être communes). Soit y la destination de $\mathcal{N}_4$ et $\mathcal{N}_6$ dans notre exemple.

Nous appelons récursivement notre algorithme de résolution des conflits (étapes 2, 3 et 4), sur les nœuds $\mathcal{N}_4$: "*then*" et $\mathcal{N}_6$: "*else*".

Nous avons un conflit d'accès à x dans le sous-terme "*then*" car le texte dérivé de $\mathcal{N}_5$ modifie x et doit (obligatoirement) être exécuté avant celui dérivé de $\mathcal{N}_4$. La seule solution au cycle formé par $\mathcal{N}_4$ et $\mathcal{N}_5$ ($\mathcal{N}_5$ avant $\mathcal{N}_4$ pour respecter la dépendance fonctionnelle, et $\mathcal{N}_4$ avant $\mathcal{N}_5$ pour éviter de dupliquer x) est que $\mathcal{N}_5$ modifie x et que $\mathcal{N}_4$ utilise une duplication de x (dans le cadre général, nous préférons faire modifier la variable conflictuelle plutôt que sa copie, pour permettre d'avoir des modifications successives d'une même variable, de manière à obtenir un flot tel qu'il est défini au chapitre 6).

Nous ajoutons donc un nœud $\mathcal{N}_7 = \mathit{duplication}(x, x_2)$ en attribut *duplications* de $\mathcal{N}_4$ racine du sous-terme *then*, qui devient $f(g(x), x_2)$. Si f est implémenté par une fonction F avec les paramètres dans le même ordre que les opérandes de f, nous obtenons comme code dérivé de $\mathcal{N}_4$:

$$([\![\mathit{duplication}(x, x_2);\ G(x);\ y := F(x, x_2);]\!], \langle\!\langle y \rangle\!\rangle)$$

Les conflits des sous-termes "*then*" et "*else*" sont résolus. Les variables utilisées et modifiées par le nœud "*if-then-else*" $\mathcal{N}_1$ (en tant que variables utilisées ou modifiées par le texte dérivé pour l'application de l'opérateur ou constructeur de $\mathcal{N}_1$) sont considérées être celles des sous-termes "*then*" et "*else*". Le nœud $\mathcal{N}_1$ utilise et modifie x sans conflit :

x est utilisée par $\mathcal{N}_1$ en $\mathcal{N}_5$ et $\mathcal{N}_7$,
x est modifiée par $\mathcal{N}_1$ en $\mathcal{N}_5$.

Nous pouvons maintenant résoudre les conflits entre $\mathcal{N}_1$ et les autres nœuds : $\mathcal{N}_2$ et $\mathcal{N}_3$. Les codes dérivés de $\mathcal{N}_1$ et $\mathcal{N}_3$ modifient tous les deux la variable v. Nous pouvons donc choisir l'un ou l'autre pour la modifier, et deux solutions (avec notre méthode) sont possibles :

Nous ajoutons donc un nœud $\mathcal{N}_8 = duplication(x, x_3)$ en attribut *duplications* de $\mathcal{N}_1$ et remplaçons x par x_3, soit dans $\mathcal{N}_3$, soit dans les sous-termes *then* et *else* :

([[$duplication(x, x_3)$;
$G(x)$;
if $B(x)$ then
begin
$duplication(x_3, x_2)$;
$G(x_3)$;
$y := F(x_3, x_2)$;
end
else
$y := H(x_3)$;
endif]],
$\langle\!\langle y \rangle\!\rangle$)
	([[$duplication(x, x_3)$;
	$G(x_3)$;
	if $B(x_3)$ then
	begin
	$duplication(x, x_2)$;
	$G(x)$;
	$y := F(x, x_2)$;
	end
	else
	$y := H(x)$;
	endif]],
	$\langle\!\langle y \rangle\!\rangle$)

■

4.2 Calcul des destinations et des affectations supplémentaires

Dans le cas du nœud $\mathcal{N}_1$ qui code le terme en partie droite de l'équation à dériver, les affectations aux paramètres formels de sortie de la procédure dérivée (quand c'est le cas) doivent être ajoutées. Dans le cas des nœuds "*if-then-else*", lorsque les résultats des sous-termes "*then*" et "*else*" ne se trouvent pas dans les mêmes variables, les affectations à des destinations uniques doivent être ajoutées.

Nous calculons pour chaque nœud la destination de ses résultats dans le domaine concret, en fonction de l'implémentation de l'opérateur en racine du terme codé par le nœud, et des destinations des opérandes. Lorsque cette destination ne correspond pas avec celle attendue (nœud $\mathcal{N}_1$, nœud "*else*" par rapport au nœud "*then*"), les affectations supplémentaires, sous forme de nœuds de post-affectation

$$duplication(variable\text{-}origine, variable\text{-}destination)$$

sont ajoutées au nœud concerné. La destination correcte du résultat après les post-affectations est placée dans l'attribut destination. Les variables opérandes après les pré-affectations sont placées dans $\mathcal{N}_i.\,In$ et les paramètres de sortie avant les post-affectations sont placées dans $\mathcal{N}_i.\,Out$

Les destinations sont calculées différemment selon les nœuds suivants :

Noeud feuille : la destination est égale à la variable que code le nœud :

$$Var(\mathcal{N}) \Rightarrow (\mathcal{N}.destination = \mathcal{N}.variable)$$

Noeud $\mathcal{N}$ codant un produit cartésien : soient $\mathtt{D}_{i\in 1\ldots n}$ les destinations calculées pour les n éléments du produit cartésien, la destination du nœud $\mathcal{N}$ est :

$$(\mathtt{D}_1.destination, \ldots, \mathtt{D}_n.destination)$$

Noeud $\mathcal{N}$ avec un opérateur constructeur ou défini : Soit $\mathcal{N}.operateur = op$ l'opérateur qui au nœud $\mathcal{N}$ est appliqué à n opérandes, et dont les destinations calculées pour les n sous-arbres sont $\mathtt{D}_{i\in 1\ldots n}$ (ou pour le produit cartésien destination d'un unique opérande dont le co-domaine de l'opérateur est un produit cartésien de sortes). Nous utilisons la règle 3.4 page 68 avec à la place de $(f_1, \ldots, f_n)$ les destinations des opérandes : $(\mathtt{D}_1, \ldots, \mathtt{D}_n)$; à la place des destinations connues $(dest_1, \ldots, dest_m)$, un produit cartésien de m valeurs égales à ϕ, et avec $(x_{n+1}, \ldots, x_{n+m})$ comme variables fraîches. Il n'y a donc pas de post-affectations.

- $f^{pre}_{i\in[1\ldots n]}$ sont attribués à $\mathcal{N}.\,In$,
- $f^{post}_{i\in[1\ldots m]}$ sont attribués à $\mathcal{N}.\,Out$ et $\mathcal{N}.\,Destination$.

Les affectations à effectuer avant la génération du texte par $\mathcal{E}(op)$ sont codées dans des nœuds $duplication(f_i, f^{pre}_i)$ placées en $\mathcal{N}.\,preaffect$.

Noeud $\mathcal{N}$ codant un "*let*" : Soient $\mathcal{N}^x$, $\mathcal{N}^v$ et $\mathcal{N}^t$ les nœuds qui codent respectivement les sous-termes "*variable*", "*valeur*" et "*terme*". Etant donné que la synthèse de la destination de chaque nœud n'introduit pas de post-affectation pour

lui-même (excepté pour $\mathcal{N}_1$ dans le cas d'une procédure avec des paramètres formels en sortie ou entrée-sortie), nous ajoutons la post-affectation suivante à $\mathcal{N}^v$: *duplication*$(\mathcal{N}^v.\,destination, \mathcal{N}^x.variable)$. Nous affectons la variable de $\mathcal{N}^x$ (qui est sa propre destination) en destination du nœud $\mathcal{N}^v$ et la destination du nœud $\mathcal{N}$ est celle de $\mathcal{N}^t$.

Une optimisation est de placer $\mathcal{N}^x.variable$ en paramètre de sortie uniquement $\mathcal{N}^v$ si celui-ci est un nœud avec opérateur et que le résultat de $\mathcal{N}^v$ est dans un tel paramètre ($\mathcal{N}^v.Out \notin \mathcal{N}^v.In$). $\mathcal{N}$ n'a pas de post-affectations.

Noeud $\mathcal{N}$ codant un "*if-then-else*" : Soient $\mathcal{N}^c$, $\mathcal{N}^t$ et $\mathcal{N}^e$ les nœuds qui codent respectivement les sous-termes "*condition*", "*then*" et "*else*" (les 3 opérandes du nœud $\mathcal{N}$). Soient $\mathtt{D}^c = \mathcal{N}^c.destinations$, $\mathtt{D}^t = \mathcal{N}^t.destinations$ et $\mathtt{D}^e = \mathcal{N}^e.destinations$, les destinations respectives des sous-termes "*condition*", "*then*" et "*else*". Dans le cas où les résultats des sous-termes "*then*" et "*else*" ne sont pas rangés dans les mêmes destinations, nous devons choisir une destination commune et ajouter les affectations supplémentaires nécessaires au sous-termes "*then*" ou "*else*".

Nous devons éviter d'avoir une destination où il y ait deux occurrences d'une même variable, et faire un minimum d'affectations. Soit $\mathtt{D} = (\mathtt{D}_1, \ldots, \mathtt{D}_m)$ la destination choisie pour le nœud *if-then-else* $\mathcal{N}$, nous devons ajouter en post-affectation de $\mathcal{N}^t$ et $\mathcal{N}^e$ les affectations nécessaires pour que ces deux nœuds aient la destination $\mathtt{D}$. Pour $\mathcal{N}^t$ par exemple, les nœuds *duplication*(x, y) ajoutés en post-affectations sont tels que :

$$\exists i \in [1 \ldots m] \cdot \begin{pmatrix} x = \mathtt{D}^t_i \\ \wedge\ y = \mathtt{D}_i \\ \wedge\ x \neq y \end{pmatrix}$$

Remarque : le nœud "*if-then-else*" n'a pas de post-affectation, et peut voir affecter son résultat à d'autres variables s'il est lui-même sous-terme "*then*" ou "*else*".

Pour obtenir une destination commune, nous pouvons :

- soit choisir $\mathtt{D}^t$ ou $\mathtt{D}^e$,
- soit utiliser une heuristique qui permette d'obtenir une destination sans multiples occurrences, qui choisisse :
  - les variables aux mêmes positions dans $\mathtt{D}^t$ et $\mathtt{D}^e$,
  - parmi les autres variables, celles qui sont des paramètres formels.

  Soient $\mathtt{D}^{t,pf}$ et $\mathtt{D}^{e,pf}$ les destinations respectives des nœuds $\mathcal{N}^t$ et $\mathcal{N}^e$ où n'apparaissent que les paramètre formels :

  $$\mathtt{D}^{t,pf}(d^{t,pf}_1, \ldots, d^{t,pf}_m) \text{ avec :}$$
  $$\forall i \in [1 \ldots m] \cdot \begin{cases} d^t_i \in Param_{In\text{-}Out} \cup Param_{Out} \Rightarrow d^{t,pf}_i = d^t_i \\ d^t_i \notin Param_{In\text{-}Out} \cup Param_{Out} \Rightarrow d^{t,pf}_i = \phi \end{cases}$$

  Le calcul est similaire pour $\mathtt{D}^{e,pf}$. Nous complétons $\mathtt{D}^{t,pf}$ avec les paramètres formels de $\mathtt{D}^{e,pf}$, qui n'apparaissent pas dans $\mathtt{D}^{t,pf}$, et ensuite avec les variables locales de $\mathtt{D}^t$:

  $$\mathtt{D} = (\mathtt{D}^{t,pf} \boxplus_L \mathtt{D}^{e,pf}) \oplus_L \mathtt{D}^t$$

ou, à partir de $\mathtt{D}^{e,pf}$:

$$\mathtt{D} = (\mathtt{D}^{e,pf} \boxplus_L \mathtt{D}^{t,pf}) \oplus_L \mathtt{D}^{e}$$

- soit effectuer une recherche exhaustive parmi tous les produits cartésiens possibles, sans multiples occurrences, constitués de variables de $\mathtt{D}^t$ ou $\mathtt{D}^e$, de sorte que les affectations à ajouter en post-affectation aux nœuds $\mathcal{N}^t$ et $\mathcal{N}^e$ soient de coût en duplications minimum.

Noeud $\mathcal{N}_1$: dans le cas où l'opérateur d'intérêt est implémenté par une fonction, le calcul des destinations de $\mathcal{N}_1$ se fait normalement comme il a été décrit plus haut.

Lorsqu'il s'agit d'une dérivation vers une procédure, nous connaissons les paramètres formels de sortie ou d'entrée-sortie de la procédure dont nous dérivons le code, qui doivent contenir une représentation des résultats algébriques en fin d'exécution de ce code. Nous noterons ces paramètres $\mathtt{d}_1, \ldots, \mathtt{d}_m$ dans l'ordre des termes algébriques qu'ils doivent représenter :

$$(\mathtt{d}_1, \ldots, \mathtt{d}_m) : (\Delta_S(s_{n+1}), \ldots, \Delta_S(s_{n+m}))$$

La destination du nœud $\mathcal{N}_1$ se fait différemment suivant les cas :

- Noeud $\mathcal{N}_1$ feuille ou produit cartésien : la destination, égale à $\mathcal{N}_1.\,Out$ se calcule tel qu'il a été décrit précédemment, puis en lui ajoutant des nœuds de post-affectation $duplication(x, y)$ tels que :

$$\exists i \in [1 \ldots m] \cdot \left(\begin{array}{l} x = \mathcal{N}_1.\,Out_i \\ \wedge\ y = \mathtt{d}_i \\ \wedge\ x \neq y \end{array} \right)$$

  La destination de $\mathcal{N}_1$ est alors égale aux paramètres formels en sortie ou entrée-sortie $(\mathtt{d}_1, \ldots, \mathtt{d}_m)$ de l'opérateur d'intérêt.
- Noeud $\mathcal{N}_1$ avec un opérateur défini : par rapport au calcul décrit précédemment, nous utilisons $(\mathtt{d}_1, \ldots, \mathtt{d}_m)$ à la place de la destination connue $(dest_1, \ldots, dest_m)$.
- Noeud $\mathcal{N}_1$ codant un "*if-then-else*" : la destination du nœud n'est pas choisie parmi celle de ses opérandes "*then*" ou "*else*" mais est fixée directement à $(\mathtt{d}_1, \ldots, \mathtt{d}_m)$.

Exemple 4.5: (suite de l'exemple 3.8 page 62).
L'équation définissant *op* est :

$$op(x, y, z, t) \Longrightarrow (concat(concat3(x, y, renverse(z)), t),\ concat(x, z),\ tri(concat(t, x)))$$

et nous avons déterminé avec le lien d'implémentation de *op* que la dérivation du terme T en partie droite de l'équation doit être un texte dont le résultat à l'exécution est rangé dans (z, x, t)

Le fichier décrivant le lien d'implémentation en bibliothèque des opérateurs utilisés dans l'équation contient les définitions suivantes :

```
Operators
 concat3     ->> CONCAT3($3:in, $2:in, $1:in): out($4)
```

```
concat      ->> CONCAT($1:in out($3), $2:in)
tri         ->> TRI($1:in): out($2)
renverse    ->> RENVERSE($1:in out($2))
```

Ce qui donne pour lien d'implémentation :

- *concat*3 reste sous forme fonctionnelle, mais l'ordre de ses paramètres est inversé par rapport à celui de l'opérateur :

$$\mathcal{E}(concat3) = [x, y, z]\{\texttt{CONCAT3(x,y,z)}\},$$
$$In(concat3) = < \pi_3, \pi_2, \pi_1 >,$$
$$Out(concat3) = < \phi, \phi, \phi >$$

- *concat* devient une procédure dont le résultat est retourné dans le premier paramètre :

$$\mathcal{E}(concat) = [x, y]\{\texttt{CONCAT(x,y)}\},$$
$$In(concat) = < \pi_1, \pi_2 >,$$
$$Out(concat) = < \pi_1, \phi >$$

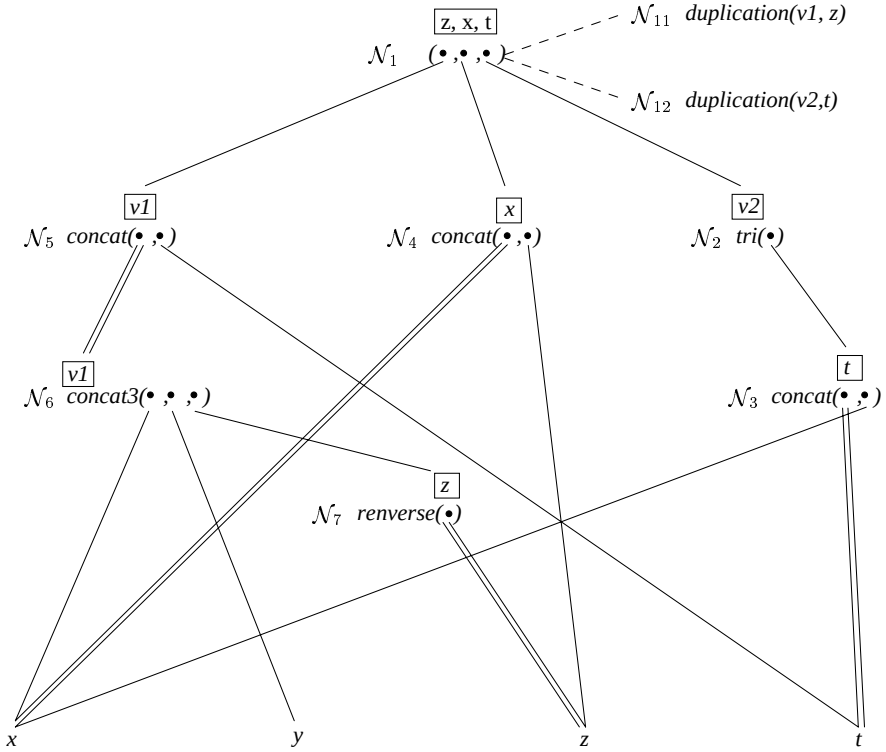
- *tri* reste sous forme fonctionnelle :

$$\mathcal{E}(tri) = [x]\{\texttt{TRI(x)}\},$$
$$In(tri) = < \pi_1 >,$$
$$Out(tri) = < \phi >$$

- *renverse* devient une procédure qui modifie son unique paramètre :

$$\mathcal{E}(renverse) = [x]\{\texttt{RENVERSE(x)}\},$$
$$In(renverse) = < \pi_1 >,$$
$$Out(renverse) = < \pi_1 >$$

Nous pouvons donc déterminer quelles sont les destinations de chaque nœud, depuis les feuilles jusqu'à la racine $\mathcal{N}_1$. Le terme T, partie droite de l'équation définissant *op* est représenté par l'arbre suivant :



Les arcs simples indiquent une utilisation d'une variable, les arcs doubles une utilisation avec modification. Les destinations calculées sont encadrées à côté de chaque nœud. Les post-affectations sont des nœuds à droite du nœud auquel ils sont liés par des arcs en pointillés.

Le codage de l'arbre abstrait est :

$\mathcal{N}_7$: $position$ 0.1.1.3 $= renverse(z)$
$\mathcal{N}_6$: $position$ 0.1.1 $= concat3(x, y, \mathcal{N}_7)$
$\mathcal{N}_5$: $position$ 0.1 $= concat(\mathcal{N}_6, t)$
$\mathcal{N}_4$: $position$ 0.2 $= concat(x, z)$
$\mathcal{N}_3$: $position$ 0.3.1 $= concat(t, x)$
$\mathcal{N}_2$: $position$ 0.3 $= tri(\mathcal{N}_3)$
$\mathcal{N}_1$: $position$ 0 $= (\mathcal{N}_5, \mathcal{N}_4, \mathcal{N}_2)$

■

4.3 Calcul des relations $\mathcal{R}_<$

Les relations $\mathcal{R}^{\vec{v}}_<(\mathcal{N}_i, \mathcal{N}_j)$ sont codées dans l'arbre abstrait par: $(i, \vec{v}) \in \mathcal{N}_j.antecedents$. Les relations $\mathcal{R}_T(\mathcal{N}_i, \mathcal{N}_j)$ sont codées dans l'arbre abstrait par: $i \in \mathcal{N}_j.operandes$. Chaque variable $v \in \vec{v}$ de $\mathcal{R}^{\vec{v}}_<(\mathcal{N}_i, \mathcal{N}_j)$ est notée $v_{(a,b)}$ avec a et b respectivement les indices d'opérandes de v dans $\mathcal{N}_i$ et $\mathcal{N}_j$, quand v a plusieurs occurrences parmi les opérandes de $\mathcal{N}_i$ ou $\mathcal{N}_j$, et avec des modes différents dans le domaine concret.

Remarques:

- Plusieurs arcs de $\mathcal{R}_<$ ne peuvent pas avoir les mêmes extrémités, c'est-à-dire que nous ne pouvons pas avoir: $\mathcal{R}^{\vec{v}}_<(\mathcal{N}_i, \mathcal{N}_j) \in \mathcal{R}_<$ et $\mathcal{R}^{\vec{w}}_<(\mathcal{N}_i, \mathcal{N}_j) \in \mathcal{R}_<$ avec $\vec{v} \neq \vec{w}$.
- Plusieurs arcs d'extrémités différentes de $\mathcal{R}_<$ peuvent être "annotés" par les mêmes variables, c'est-à-dire que nous pouvons avoir: $\mathcal{R}^{\vec{v}}_<(\mathcal{N}_i, \mathcal{N}_j) \in \mathcal{R}_<$ et $\mathcal{R}^{\vec{v}}_<(\mathcal{N}_h, \mathcal{N}_k) \in \mathcal{R}_<$ avec $i \neq h \ \vee \ j \neq k$.
- Conséquence: deux nœuds peuvent, pour résoudre des conflits sur des variables partagées différentes, devoir être évalués selon des ordres contraires, c'est-à-dire que nous pouvons avoir: $\mathcal{R}^{\vec{v}}_<(\mathcal{N}_i, \mathcal{N}_j) \in \mathcal{R}_<$ et $\mathcal{R}^{\vec{w}}_<(\mathcal{N}_j, \mathcal{N}_i) \in \mathcal{R}_<$ avec $v \neq w$.

Nous cherchons l'ensemble des relations $\mathcal{R}^{\vec{v}}_<(\mathcal{N}_i, \mathcal{N}_j)$ qui peuvent être déduites de T. Ces arcs codent des ordres d'exécution entre textes dérivés des nœuds, de manière à conserver la valeur d'un paramètre formel référencé par un nœud et désigné par son nom.

La dérivation du texte correspondant à l'application de l'opérateur en racine de chaque nœud comme cela est décrit par la règle 3.4 page 68. La correction de la dérivation implique que le texte dérivé pour chaque nœud soit appliqué à des paramètres effectifs qui représentent les valeurs des opérandes de ce nœud. Le calcul de ces paramètres effectifs se fait en partant des feuilles vers la racine du terme T (en calculant les destinations des sous-termes $t = f(t_1, \ldots, t_n)$ de T en fonction des destinations des sous-termes opérandes $t_1, \ldots, t_n$). Nous n'introduisons que des variables fraîches en paramètres effectifs de sortie uniquement. Tout conflit d'utilisation d'une variable x selon des modes différents par les codes dérivés de deux nœuds $\mathcal{N}$ et $\mathcal{N}'$ qui codent respectivement les termes t et t' a pour origine:

- soit x est un opérande de $\mathcal{N}$ (ou $\mathcal{N}'$),
- soit x est un opérande d'un sous-terme t_1 (ou t'_1) de t (ou de t') et il existe $t_1, \ldots, t_n$ (ou $t'_1, \ldots, t'_n$) tels que $t_{i\in[1,\ldots,n-1]}$ (ou $t'_{i\in[1,\ldots,n-1]}$) est opérande de l'opérateur en racine de t_{i+1} (ou t'_{i+1}) avec $t_n = t$ (ou $t'_n = t'$).

Un conflit d'utilisation de x entre le texte dérivé du nœud $\mathcal{N}$ et celui du nœud $\mathcal{N}'$ où x n'est pas un opérande, implique qu'il existe un conflit d'utilisation entre $\mathcal{N}$ et un nœud $\mathcal{N}''$ qui a x en opérande, et qui code un sous-terme t'' du terme t' codé en $\mathcal{N}'$. La résolution du conflit entre $\mathcal{N}$ et $\mathcal{N}''$ résout le conflit entre les textes dérivés de $\mathcal{N}$ et $\mathcal{N}'$. Nous ne considérons donc les conflits qu'entre nœuds où une variable est un opérande des termes codés, et est utilisée selon des modes différents par les codes endendrés. Nous ne faisons pas de résolutions de conflits entre les nœuds de post-affectation et les nœuds

de l'arbre abstrait, car un même nom de variable ne référence pas la même donnée : pour un nœud de post-affectation $duplication(x, y)$ d'un nœud $\mathcal{N}$, x représente un élément du résultat (ou le résultat entier) de l'évaluation du terme codé en $\mathcal{N}$. L'utilisation de x en opérande d'un nœud de l'arbre abstrait représente par contre un paramètre formel (ou une variable d'un "*let*"). Les nœuds de pré-affectation, avec la règle utilisée, ne font des duplications que vers des variables fraîches, nous ne faisons donc pas de résolution de conflits entre les nœuds de pré-affectation et les nœuds de l'arbre abstrait. Nous résolvons les conflits entre nœuds de post-affectation d'un même nœud, qui peuvent exister pour les nœuds qui codent les sous termes "*then*" et "*else*" d'un "*if-then-else*", ou pour $\mathcal{N}_1$.

Remarque : la résolution des conflits peut modifier la destination de toute une séquence de nœuds qui sont successivement chacun l'opérande du suivant, et qui ont un (ou des) paramètre(s) en entrée-sortie. Le choix d'une destination commune aux sous-termes "*then*" et "*else*" d'un terme "*if-then-else*", doit à nouveau être appliqué après la résolution des conflits et ne doit pas remettre en cause cette résolution. Nous considérons donc les conflits entre nœuds de l'arbre abstrait en fonction de leurs variables qui sont des paramètres formels de l'opérateur d'intérêt ou des variables d'un "*let*" :

- utilisées par :
  - les pré-affectations,
  - le texte dérivé pour appliquer l'opérateur du nœud dans le domaine concret.
- modifiées par :
  - le texte dérivé pour appliquer l'opérateur du nœud dans le domaine concret,
  - les post-affectations.

Les étape suivantes de la résolution des conflits d'accès aux variables sont appliquées à un graphe G, qui code un terme algébrique T et les dépendances fonctionnelles et opérationnelles entre nœuds du graphe. La construction du graphe qui code le terme en partie droite de l'équation à dériver est décrite en section 4.1.5 page 77, mais la résolution des conflits peut être appliquée à un graphe qui en code un sous-terme. Le nœud qui représente le terme T par G est $\mathcal{N}_{Init}$ ($\mathcal{N}_1$ lorsque T est le terme en partie droite de l'équation à dériver).

Le calcul des relations $\mathcal{R}_<$ se fait par un parcours de l'arbre en profondeur d'abord. Pour chaque nœud et chaque feuille (nœud codant une variable) $\mathcal{N}_i$ sont calculées les destinations, les pré- et post-affectations, et les attributs suivants :

`U` la liste des variables utilisées (et éventuellement modifiées) dans le sous-arbre dont $\mathcal{N}_i$ est la racine. Chaque utilisation code un nœud et la liste des variables qu'il utilise avec leurs positions d'opérandes.

`M` la liste des variables modifiées (et éventuellement utilisées) dans le sous-arbre dont $\mathcal{N}_i$ est la racine. Chaque modification code un nœud et la liste des variables qu'il modifie avec leurs positions d'opérandes.

Nous notons pour $\mathtt{U}$ et $\mathtt{M}$ une liste de doublets, tel $(\vec{v}, \mathcal{N}_i)$ lorsque l'opérateur du nœud $\mathcal{N}_i$ utilise ou modifie les variables $\vec{v}$ de ses paramètres. Nous notons le doublet $(\vec{v}, (\mathcal{N}_i, \mathcal{N}_j))$ pour une relation $\mathcal{R}^{\vec{v}}_{<}(\mathcal{N}_i, \mathcal{N}_j)$.

Le calcul des attributs se fait de la manière suivante selon les nœuds :

Une feuille :

$$\mathtt{U} = \{\,\}, \mathtt{M} = \{\,\}.$$

Un nœud $\mathcal{N}$ de pré- ou post-affectation *duplication*(x, y) **:**

$$\mathtt{U} = \{((x), \mathcal{N})\}, \mathtt{M} = \{((y), \mathcal{N})\}.$$

Un nœud $\mathcal{N}_o$ de constructeur ou opérateur défini *op* **:**

soient les attributs calculés pour

- ses p pré-affectations : $\mathtt{U}'_{i\in[1..p]}$ et $\mathtt{M}'_{i\in[1..p]}$,
- ses n opérandes : $\mathtt{U}_{i\in[1..n]}$ et $\mathtt{M}_{i\in[1..n]}$,
- ses q post-affectations $\mathtt{U}''_{i\in[1..q]}$ et $\mathtt{M}''_{i\in[1..q]}$.

Nous résolvons d'abord les conflits qui peuvent exister dans les post-affectations (pour $x, y := v, x$ par exemple, et même avec un cycle pour $x, y := y, x$). L'ensemble $\mathtt{R}$ de leurs relations $\mathcal{R}_<$ est calculé par :

$$\mathtt{R} = \left\{ ((v), (\mathcal{N}_a, \mathcal{N}_b)) \;\middle|\; \begin{array}{l} \exists j, k \in [1 \ldots q] \wedge j \neq k \\ \wedge\ ((v), \mathcal{N}_a) \in \mathtt{U}''_j \\ \wedge\ ((v), \mathcal{N}_b) \in \mathtt{M}''_k \end{array} \right\}$$

Nous appliquons donc les étapes 2 et 3 de notre algorithme sur le graphe formé uniquement des arcs de $\mathtt{R}$.

Le nœud $\mathcal{N}_o$ a ses attributs $\mathtt{U}$ et $\mathtt{M}$ synthétisés par :

$$\mathtt{U} = \bigcup_{i\in 1\ldots n} \mathtt{U}_i \cup \left\{ (\vec{v}, \mathcal{N}_o) \;\middle|\; \forall v \in \vec{v} \cdot \begin{pmatrix} v \in \mathcal{N}_o.operandes \\ \wedge\ Var(v) \\ \wedge\ v \in Param_{In} \cup Param_{In\text{-}Out} \end{pmatrix} \right\},$$

les variables de $\vec{v}$ sont les opérandes de $\mathcal{N}_o$ qui sont des paramètres formels en entrée ou entrée-sortie de l'opérateur d'intérêt, ou des variables définies par un "*let*".

$$\mathtt{M} = \bigcup_{i\in 1\ldots n} \mathtt{M}_i \cup \left\{ (\vec{v}, \mathcal{N}_o) \;\middle|\; \forall v \in \vec{v} \cdot \begin{pmatrix} v \in \mathcal{N}_o.operandes \\ \wedge\ Var(v) \\ \wedge\ v \in P^{op}_{In\text{-}Out}(\mathcal{N}_o.operandes) \\ \wedge\ v \in Param_{Out} \cup Param_{In\text{-}Out} \end{pmatrix} \right\},$$

les variables de $\vec{v}$ sont les opérandes de $\mathcal{N}_o$ qui sont des paramètres formels en sortie ou entrée-sortie de l'opérateur d'intérêt, ou des variables définies par un "*let*".

Un nœud $\mathcal{N}_o$ dont l'opérateur prédéfini est *if-then-else* :

Les termes *then* et *else* et la condition booléenne ont un ordre de calcul qui n'est pas quelconque. Le nœud est traité comme si son opérateur utilisait et modifiait les variables utilisées et modifiées dans ses 2 sous-arbres *then* et *else*. Nous pouvons exécuter le bloc condition à l'extérieur du code dérivé du *if-then-else*. La résolution des conflits se fait en plusieurs étapes :

- nous résolvons d'abord les conflits dans les graphes qui codent les termes *then* et *else*,
- nous calculons les destinations communes du *if-then-else* à partir des destinations des nœuds *then* et *else* qui ont pu changer avec la résolution des conflits, et modifions les post-affectations des nœuds *then* et *else* en conséquence. Ces destinations sont également celles du nœud *if-then-else*,
- nous résolvons les conflits qui ont pu être créés dans les nouvelles post-affectations des nœuds *then* et *else*, de manière locale, comme cela est fait pour les post-affectations d'un nœud qui possède un constructeur ou opérateur défini,
- nous calculons les utilisations et modifications du nœud *if-then-else* de la manière suivante : soient $\mathtt{U}_1$, $\mathtt{U}_2$ et $\mathtt{U}_3$ les attributs calculés respectivement pour les nœuds *then* et *else*, nous calculons :

$$\mathtt{U} = \mathtt{U}_1 \cup \{(\vec{v}, \mathcal{N}_o) \mid \exists i \in [2;3] \ \cdot (\vec{v}, \mathcal{N}_j) \in \mathtt{U}_i\}$$

$$\mathtt{M} = \mathtt{M}_1 \cup \{(\vec{v}, \mathcal{N}_o) \mid \exists i \in [2;3] \ \cdot (\vec{v}, \mathcal{N}_j) \in \mathtt{M}_i\}$$

L'ensemble des relations $\mathcal{R}_<$ recherchées dans tout le graphe G est obtenu en calculant l'ensemble $\mathtt{R}$ suivant pour les utilisations $\mathtt{U}$ et modifications $\mathtt{M}$ au nœud $\mathcal{N}_{Init}$:

$$\mathtt{R} = \left\{ (\vec{v}, (\mathcal{N}_a, \mathcal{N}_b)) \;\middle|\; \begin{array}{l} \exists j,k \in [1 \ldots n] \wedge j \neq k \\ \wedge \ (\vec{v'}, \mathcal{N}_a) \in \mathtt{U}_j \\ \wedge \ (\vec{v''}, \mathcal{N}_b) \in \mathtt{M}_k \\ \wedge \ \vec{v} = \vec{v'} \cap \vec{v''} \neq \phi \end{array} \right\}$$

Exemple 4.6: (suite de l'exemple 4.5 page 87).

Nous obtenons par l'application de l'algorithme en synthétisant les attributs des feuilles vers la racine $\mathcal{N}_1$, la liste des doublets (variable utilisées , nœud utilisateur) suivante :

$$\mathcal{N}_1.\mathtt{U} = [((x,z),\mathcal{N}_4); ((x,y),\mathcal{N}_6); ((t,x),\mathcal{N}_3); ((t),\mathcal{N}_5); ((z),\mathcal{N}_7)]$$

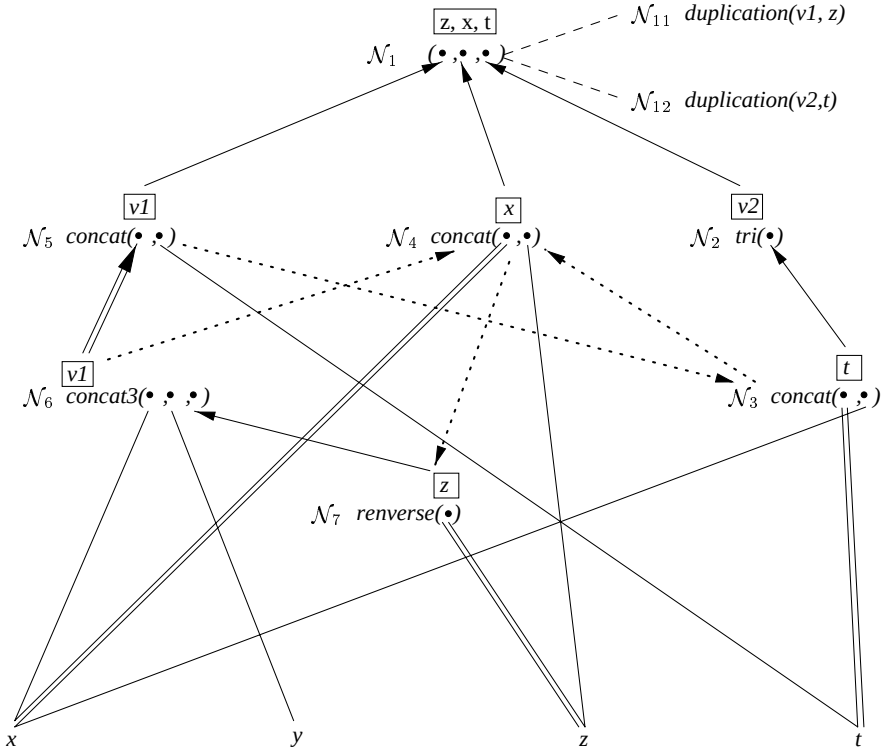
La liste des doublets (variables modifiées , nœud modificateur) :

$$\mathcal{N}_1.\mathtt{M} = [(z,\mathcal{N}_7); (x,\mathcal{N}_4); (t,\mathcal{N}_3)]$$

Et la liste des arcs de $\mathcal{R}_<$ qui doivent le plus possible être respecté lors de l'évaluation de l'arbre dans le domaine concret :

$$\begin{aligned} \mathcal{R}_< &= \mathcal{N}_1.\mathtt{R} \\ &= [(x,(\mathcal{N}_6,\mathcal{N}_4)); (t,(\mathcal{N}_5,\mathcal{N}_3)); (z,(\mathcal{N}_4,\mathcal{N}_7)); (x,(\mathcal{N}_3,\mathcal{N}_4))] \end{aligned}$$

Le graphe formé par l'ensemble des arcs $\mathcal{R}_<$ et $\mathcal{R}_T$ forme le graphe suivant, où tout nœud origine d'un arc doit, dans le domaine concret être évalué avant le nœud à son extrémité pour que chaque variable paramètre effectif ait bien la valeur qu'elle doit représenter à chacune de ses utilisations:



Les arcs orientés indiquent l'ordre d'évaluation entre nœuds à respecter. Les arcs "en pointillés" sont les arcs de $\mathcal{R}_<$.

■

4.4 Calcul des cycles dans les relations $\mathcal{R}_<$ et $\mathcal{R}_T$

Dans cette section et la suivante, nous allons montrer un algorithme de recherche puis de suppression de cycles dans un graphe. De tels algorithmes sont très connus [Ber73, Sak84a, Sak84b, DP90] mais ceux que nous montrons exploitent les propriétés particulières des graphes que nous manipulons.

Pour savoir si, pour calculer T dans le domaine concret, nous pouvons évaluer les nœuds de l'arbre abstrait en respectant à la fois les relations $\mathcal{R}_T$ et les relations $\mathcal{R}_<$, nous devons vérifier qu'il n'existe pas de cycle dans $\mathcal{R}_< \cup \mathcal{R}_T$. Nous allons donc chercher les cycles hamiltoniens (passant une seule fois par chaque sommet) qui peuvent exister dans le graphe défini par les arcs de $\mathcal{R}_*$.

Remarque: les cycles ont été créés par l'ajout de certains arcs de $\mathcal{R}_<$ à ceux de l'arbre abstrait T. Les arcs $\mathcal{R}_T$ ne peuvent être supprimés puisqu'ils représentent les dépendances fonctionnelles entre sous-termes algébriques. Nous allons donc nous restreindre à chercher pour chaque arc de $\mathcal{R}_<$ l'ensemble des cycles auxquels il appartient.

Soit $\mathcal{K}$ l'ensemble des cycles existants dans l'ensemble $\mathcal{R}_*$. Soit $Cycles_r()$ la fonction qui pour tout graphe passé en argument, renvoie tous ses cycles qui comportent l'arc r.

$$\mathcal{K} = \bigcup_{r \in \mathcal{R}_<} Cycles_r(\mathcal{R}_*)$$

Nous fixerons un ordre quelconque des arcs dans l'ensemble $\mathcal{R}_<$. Les ensembles calculés par $Cycles_r(\mathcal{R}_*)$ et $Cycles_{r'}(\mathcal{R}_*), r \neq r'$ ne sont pas toujours disjoints. Pour ne pas re-calculer un même cycle, après chaque calcul d'un ensemble de cycles passant par un arc R, nous ôtons cet arc du graphe dans lequel nous recherchons d'autres cycles. Nous aurons donc :

$$\mathcal{K} = \bigcup_{\substack{i \in 1 \ldots Card(\mathcal{R}_<) \\ r_i \in \mathcal{R}_<}} Cycles_{r_i}(\mathcal{R}_* \setminus \{r_1 \ldots r_{i-1}\})$$

Soit $Chemins_{\mathcal{R}}^{\mathcal{N}'}()$ la fonction qui, appliquée à un nœud $\mathcal{N}$ renvoie l'ensemble des chemins (un chemin est une séquence d'arcs continus) du graphe $\mathcal{R}$ qui à partir de $\mathcal{N}$ atteignent $\mathcal{N}'$:

$$\begin{aligned} \mathcal{K} &= \bigcup_{\mathcal{R}(\mathcal{N},\mathcal{N}') \in \mathcal{R}_<} Cycles_{\mathcal{R}(\mathcal{N},\mathcal{N}')}(\mathcal{R}_*) \\ &= \bigcup_{\mathcal{R}(\mathcal{N},\mathcal{N}') \in \mathcal{R}_<} \left\{ C; \mathcal{R}(\mathcal{N},\mathcal{N}') \mid C \in Chemins_{\mathcal{R}_* - \mathcal{R}(\mathcal{N},\mathcal{N}')}^{\mathcal{N}}(\mathcal{N}') \right\} \end{aligned}$$

$$\begin{aligned} Chemins_{\mathcal{R}}^{\mathcal{N}'}(\mathcal{N}') = &\{\{\mathcal{R}(\mathcal{N},\mathcal{N}')\} \mid \mathcal{R}(\mathcal{N},\mathcal{N}') \in \mathcal{R}\} \\ &\cup \left\{ C; \mathcal{R}(\mathcal{N},\mathcal{N}'') \mid \mathcal{R}(\mathcal{N},\mathcal{N}'') \in \mathcal{R} \right. \\ &\qquad \left. \wedge\, C \in Chemins_{\mathcal{R} - \mathcal{R}(\mathcal{N},\mathcal{N}'')}^{\mathcal{N}'}(\mathcal{N}'') \right\} \end{aligned}$$

Exemple 4.7: (suite de l'exemple 4.6 page 93).
Les cycles obtenus, indexés par les arcs de :

$$\mathcal{R}_< = [(x, (\mathcal{N}_6, \mathcal{N}_4)); (t, (\mathcal{N}_5, \mathcal{N}_3)); (z, (\mathcal{N}_4, \mathcal{N}_7)); (x, (\mathcal{N}_3, \mathcal{N}_4))]$$

sont :

$$\mathcal{K} = [\{[(\mathcal{N}_6, \mathcal{N}_4); (\mathcal{N}_4, \mathcal{N}_7); (\mathcal{N}_7, \mathcal{N}_6)]\}; \{[(\mathcal{N}_5, \mathcal{N}_3); (\mathcal{N}_3, \mathcal{N}_4); (\mathcal{N}_4, \mathcal{N}_7); (\mathcal{N}_7, \mathcal{N}_6); (\mathcal{N}_6, \mathcal{N}_5)]\}; \{\}; \{\}]$$

Le premier élément de $\mathcal{K}$ est l'ensemble de tous les cycles qui passent par le premier élément de $\mathcal{R}_<$ (l'arc qui code $\mathcal{R}^x_<(\mathcal{N}_6, \mathcal{N}_4)$). Un seul cycle passe par l'arc qui code $\mathcal{R}^x_<(\mathcal{N}_6, \mathcal{N}_4)$. Une fois cet arc supprimé, un seul cycle passe par $(\mathcal{N}_5, \mathcal{N}_3)$, et il ne reste ensuite plus aucun cycle (s'il y avait encore, ils passeraient par $(\mathcal{N}_4, \mathcal{N}_7)$ ou $(\mathcal{N}_3, \mathcal{N}_4)$).
■

4.5 Calcul des coupures des cycles

Nous avons un graphe orienté acyclique G qui code un terme T, et dont les arcs représentent les ordres à respecter dans l'évaluation des sous-termes de T dans le domaine concret (relations $\mathcal{R}_T$ entre nœuds). T est représenté par le nœud $\mathcal{N}_{Init}$ ($\mathcal{N}_1$ lorsque T est le terme en partie droite de l'équation qui définit l'opérateur d'intérêt). Nous avons également un ensemble d'arcs orientés entre les nœuds de G qui représentent des relations $\mathcal{R}^{\vec{v}}_<(\mathcal{N}_i, \mathcal{N}_j)$ (ce qui signifie que le texte dérivé pour appliquer l'opérateur du nœud $\mathcal{N}_i$ dans le domaine concret utilise les variables $\vec{v}$, et que celui dérivé pour $\mathcal{N}_j$ les modifie. Les variables $\vec{v}$ sont des paramètres formels de l'opérateur d'intérêt ou définies par un "*let*").

Pour effectuer le calcul de T dans le domaine concret, il faut respecter strictement les relations $\mathcal{R}_T$ et au maximum les relations $\mathcal{R}_<$. Lorsque l'ensemble des arcs qui représentent ces relations contient un cycle, cela signifie qu'un ordre ne pourra pas être respecté. Il faut donc couper les cycles existants dans l'union de ces 2 ensembles. Etant donné qu'un arc défini par une relation $\mathcal{R}_T$ ne peut être supprimé (nous n'engendrons pas de code pour un "langage à évaluation paresseuse", un opérateur n'est appliqué que sur le résulat de l'évaluation de ses opérande), une coupure d'un cycle ne peut se faire que par la suppression d'un de ses arcs défini par une relation $\mathcal{R}_<$.

La méthode de recherche des coupures de cycles $\mathcal{K}$ parmi l'ensemble $\mathcal{R}_<$ est semi exhaustive. Récursivement pour un ensemble R de coupures potentielles (initialement $\mathcal{R}_<$), nous cherchons s'il existe des ensembles de coupures dans l'ensemble égal à R moins un de ses éléments r, pour l'ensemble de cycles K (initialement $\mathcal{K}$) auquel nous avons appliqué la coupure r (supprimé tous les cycles qui contiennent r) :

- s'il existe des ensembles de coupures pour $K - \{k \in K \mid r \in k\}$ alors chacun de ces ensembles, auquel nous ajoutons r est un ensemble de coupures de K. Nous pouvons continuer à chercher par la même méthode s'il existe des ensembles de coupures des cycles de K parmi $R - \{r\}$,
- S'il n'existe pas de coupures de cycles de $K - \{k \in K \mid r \in k\}$ parmi $R - \{r\}$, alors il n'en existera pas de K parmi $R - \{r\}$.

La condition d'arrêt de la récursion est :

- soit la recherche d'un ensemble de coupures d'un ensemble K vide, pour lequel l'ensemble vide est un ensemble de coupures solution,
- soit la recherche d'un ensemble de coupures d'un ensemble K non vide, parmi un ensemble R vide. Il n'y a dans ce cas pas de solution.

La suppression d'un arc $r \in \mathcal{R}_<$ est un conflit d'utilisation de variables qui ne peut être réglé par un ordre d'exécution des textes dérivés de deux nœuds (texte utilisateur avant texte modificateur). Cette coupure représente donc une (ou des) duplication(s) de variable(s) avant modification pour que la (ou les) valeur(s) dupliquée(s) soi(en)t utilisée(s) ultérieurement. Nous choisissons donc parmi tous les ensembles de coupures possibles, celui dont le coût total en duplications est le minimum. Nous donnerons le coût en duplications d'un ensemble de coupures, que calcule fonction *Cout*, dans la section 4.6 suivante.

Nous avons des ensembles $\mathcal{K}_{i\in[1...Card(\mathcal{R}_<)]}$ qui correspondent à tous les cycles contenant $r_{i\in[1...Card(\mathcal{R}_<)]} \in \mathcal{R}_<$ moins les cycles appartenant aux $\mathcal{K}_{j\in[1...i-1]}$. Les cycles de $\mathcal{K}_i$ sont par définition connexes par $r_{i\in[1...Card(\mathcal{R}_<)]} \in \mathcal{R}_<$. Chaque arc $r_i \in \mathcal{R}_<$ est donc une coupure de tous les cycles de $\mathcal{K}_i$, et des cycles des ensembles $\mathcal{K}_{j,\ j\in[1...i-1]}$ qui le contiennent.

Soit une fonction $Coupures(K, R)$ qui, pour un ensemble K d'ensembles de cycles renvoie l'ensemble des arcs de R dont la soustraction des cycles de K supprime ces cycles, et dont le coût de la duplication des variables associées est minimum.
Pour déterminer l'ensemble $\mathcal{J}$ des coupures de coût minimal des cycles $\mathcal{K}_{i\in[1...Card(\mathcal{R}_<)]}$ de $\mathcal{R}_*$ parmi les arcs de $\mathcal{R}_<$, nous devons calculer :

$$\begin{aligned}\mathcal{J} &= Coupures(\mathcal{K}, \mathcal{R}_<) \\ &= Coupures((C_1 \dots C_n), (r_1 \dots r_n)),\end{aligned}$$
$$\begin{aligned} n &= Card(\mathcal{R}_<) \\ \mathcal{K} &= (C_1 \dots C_n) \\ \mathcal{R}_< &= (r_1 \dots r_n)\end{aligned}$$

L'algorithme de la fonction $Coupures(K, R)$ consiste en un calcul récursif des ensembles $J_{i\in[1...Card(R)]}$ des coupures de coût minimal des cycles de K après avoir supprimé les cycles contenant $r_i \in R$, et un choix de l'ensemble de coupures parmi $J_i \cup \{r_i\}$ qui est de coût minimal.

$\mathcal{J}_{i\in[1...Card(\mathcal{R}_<)]}$ est calculé de la manière suivante :

$$\mathcal{J} = Coupures(\mathcal{K}, \mathcal{R}_<) = \underset{i\in[1...Card(\mathcal{R}_<)]}{Min}(\mathcal{J}_i)$$

$$\mathcal{J}_i = \{r_i\} \cup Coupures(\ (\bigcup_{j\in[1...Card(\mathcal{R}_<)]}(\mathcal{K}_j) \setminus \mathcal{K}_i - \{C \in \bigcup_{j\in[1...i-1]}(\mathcal{K}_j) \mid r_i \in C\}\ ),$$
$$\forall i \in [1 \dots Card(\mathcal{R}_<)]\ \forall r_i \in \mathcal{R}_<)$$

Nous pouvons donc définir la fonction de coupure $Coupures(K, R)$ qui calcule l'ensemble de coût minimal d'arcs de R qui permet de couper les cycles de K.

$$Coupures(K, R) = \underset{r \in R}{Min} \quad \{\{r\} \cup Coupures(K - \{C \in \mathcal{K} | r \; coupe \; C\}, R - \{r\})\}$$

Pour ne pas recalculer plusieurs fois le même ensemble de coupures, nous allons ôter l'arc dont nous venons de calculer l'ensemble $Coupures_R$, des arcs utilisés pour les calculs des ensembles de coupure suivants :

$$Coupures(K, R) = Inf \left(\begin{array}{l} \{r\} \cup Coupures(K - \{C \in K | r \; coupe \; C\}, R - \{r\}), \\ Coupures(K, R - \{r\}) \end{array} \right)_{\forall r \in R}$$

Un arc coupe un cycle hamiltonien s'il fait partie de ce cycle. Et nous allons utiliser le fait de n'avoir pas à tester si $r_i \in \mathcal{K}_j$ quand $i > j$:

$$Coupures \left(\begin{array}{l} K = \{C_{a_1}, \ldots, C_{a_m}, C_{(a_m)+1}, \ldots, C_{(a_m)+n}\}, \\ R = \{r_{a_m}, r_{(a_m)+1}, \ldots, r_{(a_m)+n}\} \end{array} \right) =$$

$$Inf \left(\begin{array}{l} \{r_{a_m}\} \cup Coupures(\{C_{i>a_m} \in K\} \cup \{C_{i<a_m} \in K | r \notin C\}, R - \{r_{(a_m)}\}), \\ Coupures(K, R - \{r_{(a_m)}\}) \end{array} \right)$$

La fonction Inf est paramétrée par la fonction $Cout$.

Nous notons $\mathcal{J}$ l'ensemble des coupures de coût minimum.

$$\begin{array}{ll} \mathcal{J} = Coupures(\mathcal{R}) & = Coupures(\mathcal{K}, \mathcal{R}_<) \\ & = Coupures(C_1 \ldots C_n \in \mathcal{K}, r_1 \ldots r_n \in \mathcal{R}_<), \\ & n = Card(\mathcal{R}_<) \end{array}$$

Nous changeons de profil pour la fonction $Coupures$, de manière à bénéficier de l'avantage à coder les ensembles K et E dont les élément sont indicés, par des liste (avec les opérateurs classiques *first*, *tail* et :: pour la construction) :

$$Coupures \left(\begin{array}{l} K = \{C_{a_1}, \ldots, C_{a_m}, C_{(a_m)+1}, \ldots, C_{(a_m)+n}\}, \\ R = \{r_{a_m}, r_{(a_m)+1}, \ldots, r_{(a_m)+n}\} \end{array} \right) =$$

$$Coupures' \left(\begin{array}{l} K_{pre-R} = \{C_{i<a_m}\} \\ K_{post-R} = \{C_{i \geq a_m}\} \\ R = \{r_{a_m}, r_{(a_m)+1}, \ldots, r_{(a_m)+n}\} \end{array} \right)$$

Nous obtenons :

$$Coupures'(K_{pre-R}, K_{post-R}, R) =$$

$$Inf \left(\begin{array}{l} fst(R) :: Coupures'(\{C \in K_{pre-R} | first(R) \notin C\}, tail(K_{post-R}), tail(R)), \\ Coupures'(first(K_{post-R}) :: K_{pre-R}, tail(K_{post-R}), tail(R)) \end{array} \right)$$

Et l'ensemble $\mathcal{J}$ des coupures de coût minimum est calculé par :

$$\begin{array}{ll} \mathcal{J} = & = Coupures(\mathcal{K}, \mathcal{R}_<) \\ & = Coupures'([\,], \mathcal{K}, \mathcal{R}_<) \end{array}$$

Exemple 4.8: (suite de l'exemple 4.7 page 96).

Pour simplifier, sans différencier le coût de duplication de deux variables, si nous prenons pour critère de choix entre 2 listes d'arcs à couper, la longueur plus courte et la première liste en cas de listes de longueurs egales, nous obtenons l'arc :

$(\mathcal{N}_4, \mathcal{N}_7)$

qui en effet coupe les 2 cycles :

$$[\,[(\mathcal{N}_6,\mathcal{N}_4);(\mathcal{N}_4,\mathcal{N}_7);(\mathcal{N}_7,\mathcal{N}_6)];\ [(\mathcal{N}_5,\mathcal{N}_3);(\mathcal{N}_3,\mathcal{N}_4);(\mathcal{N}_4,\mathcal{N}_7);(\mathcal{N}_7,\mathcal{N}_6);(\mathcal{N}_6,\mathcal{N}_5)]\,]$$

■

4.6 Application des coupures de cycles

Soit G' le graphe G (constitué des nœuds $\mathcal{N}_i$ et des arcs de $\mathcal{R}_T$) auquel nous ajoutons les arcs de $\mathcal{R}_<$. S'il existe des cycles, nous devons supprimer de G' les arcs de l'ensemble $\mathcal{J}$ des coupures pour obtenir un graphe G'' orienté sans cycle (d.a.g.). Les ordres entre nœuds codés par les arcs de G'' peuvent être respectés pour évaluer la valeur du terme représenté par chaque nœud, dans le domaine concret.

Une coupure $\mathcal{R}^{\vec{v}}_<(\mathcal{N}_i, \mathcal{N}_j)$ signifie que le nœud $\mathcal{N}_j$ modificateur des variables $\vec{v}$ doit être exécuté avant le nœud $\mathcal{N}_i$ qui utilise les valeurs des variables $\vec{v}$ avant leur modification. Des duplications de sauvegarde doivent donc être effectuées. Nous avons 2 cas :

Le graphe G est un ensemble de post-affectations d'un même nœud $\mathcal{N}$:

il n'y a qu'une seule occurrence de chaque variable parmi toutes les variables auxquelles nous affectons une valeur, ou dont nous affectons la valeur, dans les nœuds de post-affectation (conséquence des propriétés des combinateurs *In* et *Out* de chaque opérateur du graphe et de l'opérateur d'intérêt). Quel que soit l'arc $\mathcal{R}^x_<(\mathcal{N}, \mathcal{N}')$ choisi comme coupure, (il peut y en avoir plusieurs), il est le seul dont la variable conflictuelle est x. Nous avons donc $\mathcal{N} = \mathit{duplication}(x, y)$ et $\mathcal{N}' = \mathit{duplication}(z, x)$, ($y$ peut être égal à z). Nous supprimons donc l'arc $\mathcal{R}^x_<(\mathcal{N}, \mathcal{N}')$, nous insérons parmi les nœuds de post-affectation un nœud $\mathcal{N}'' = \mathit{duplication}(x, v)$ avec v une variable fraîche, nous remplaçons la variable x dans le nœud $\mathcal{N}$ qui devient $\mathit{duplication}(v, y)$ et ajoutons l'arc $\mathcal{R}^x_<(\mathcal{N}'', \mathcal{N}')$.

Le coût de la suppression d'un arc $\mathcal{R}^x_<(\mathcal{N}, \mathcal{N}')$ de post-affectations est donc le coût de la duplication de x (que l'on peut approximer par la taille mémoire moyenne occupée par le codage en Ada d'une donnée du type qui implémente la sorte de x).

Le graphe G code un terme T :

Nous devons supprimer des arcs $\mathcal{R}^{\vec{v}}_<(\mathcal{N}, \mathcal{N}')$ de sorte que le texte dérivé de $\mathcal{N}'$ puisse être exécuté avant celui de $\mathcal{N}$, sans que $\mathcal{N}$ utilise des valeurs incorrectes pour ses opérandes (des paramètres formels de l'opérateur d'intérêt ou des variables définies par un "*let*" qui seraient modifiées auparavant par l'exécution du texte dérivé de $\mathcal{N}$).

Nous pouvons avoir plusieurs coupures dont les variables conflictuelles contiennent la même variable : soit par exemple, parmi un ensemble de coupures : $\mathcal{R}^{\vec{v}}_{<}(\mathcal{N}_1, \mathcal{N}_2)$ et $\mathcal{R}^{\vec{w}}_{<}(\mathcal{N}_3, \mathcal{N}_4)$ avec $x \in \vec{v}$ et $x \in \vec{w}$. La variable x est utilisée (et éventuellement modifiée) par $\mathcal{N}_1$ et $\mathcal{N}_3$. La variable x est modifiée par $\mathcal{N}_2$ et $\mathcal{N}_4$ (et utilisée, non uniquement modifiée dans notre première approche, où seuls les nœuds qui codent un sous terme "*then*" ou un sous terme "*else*" peuvent avoir une destination non calculée à partir de leurs propres opérandes, et avoir une variable en mode sortie uniquement). Nous devons donc exécuter le texte dérivé de $\mathcal{N}_2$ avant celui dérivé de $\mathcal{N}_1$, et celui de $\mathcal{N}_4$ avant celui de $\mathcal{N}_3$.

L'ordre partiel défini par ces deux ordres est compatible avec six ordres totaux sur $\mathcal{N}_1, \mathcal{N}_2, \mathcal{N}_3, \mathcal{N}_4$, qui sont déterminés par leur position dans le graphe, et le choix de l'ordre d'évaluation des sous-termes (profondeur ou largeur d'abord, de gauche à droite ou inversement parmi les opérandes).

De plus, si nous avons ces deux coupures, cela signifie que nous avons également, soit $\mathcal{R}^{\vec{x}}_{<}(\mathcal{N}_1, \mathcal{N}_4)$, soit $\mathcal{R}^{\vec{x}}_{<}(\mathcal{N}_3, \mathcal{N}_2)$ qui ne peut être respecté, et qui fait partie de l'ensemble des coupures.

Nous voyons que :

- nous ne pouvons faire une simple duplication des variables $\vec{w}$ pour chaque coupure $\mathcal{R}^{\vec{w}}_{<}$, comme c'est le cas pour un ensemble de post-affectations,
- nous ne pouvons connaître dans certains cas le nœud modificateur dont le texte dérivé est exécuté le premier si l'ordre d'évaluation n'est pas fixé.

Nous proposons donc pour chaque variable $v \in \vec{v}$ telle que $\mathcal{R}^{\vec{v}}_{<}(\mathcal{N}, \mathcal{N}')$ est une coupure, de :

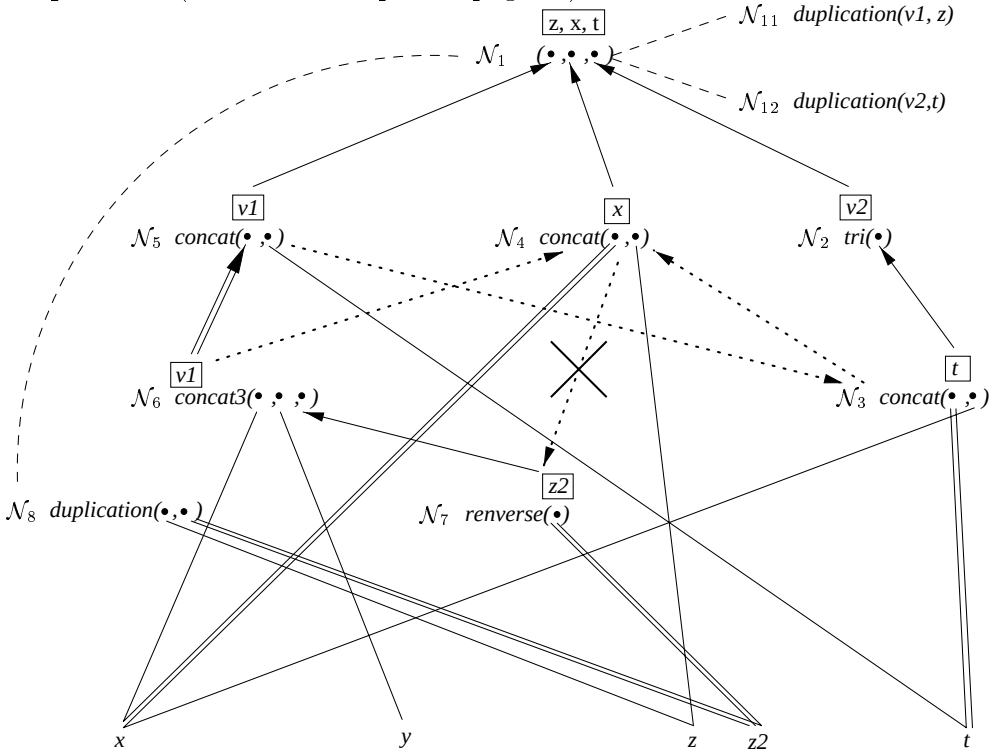
- choisir parmi l'ensemble des n coupures $\mathcal{R}^{\vec{w}_i}_{<}(\mathcal{N}_i, \mathcal{N}'_i)_{i\in[1\ldots n]}$ telles que $v \in \vec{w}_i$, le nœud $\mathcal{N}'_{j\in[1\ldots n]}$ dont le texte dérivé modifie réellement la variable x,
- pour chaque nœud $\mathcal{N}_{i\in[1\ldots n]}$ ou $\mathcal{N}'_{i\in[1\ldots n]\neq j}$ dont le texte dérivé modifie x,
  - créer un nœud $duplication(x, v)$ avec v une variable fraîche, en nœuds "sauvegardes" de $\mathcal{N}_{Init}$ du graphe G pour lequel nous résolvons les conflits,
  - remplacer x par v dans $\mathcal{N}_i$ (ou $\mathcal{N}'_i$) à sa position d'opérande modifié.
- si un nœud $\mathcal{N}_{i\in[1\ldots n]}$ au moins utilise x sans le modifier, ajouter un nœud duplication (x, v) avec v une variable fraîche, aux nœuds "sauvegardes" de $\mathcal{N}_{Init}$, et remplacer x par v dans tous les nœuds $\mathcal{N}_{i\in[1\ldots n]}$ dont le texte dérivé utilise x sans la modifier,
- supprimer les arcs $\mathcal{R}^{\vec{w}}_{<}(\mathcal{N}, \mathcal{N}')$ coupures de l'ensemble des arcs $\mathcal{R}_{<}$.

Le remplacement d'une variable x par une autre se fait :

- pour un nœud qui code l'application d'un constructeur ou opérateur défini : en remplaçant le nœud opérande qui code la variable x par un nœud opérande qui code la variable v
- pour un nœud qui code un "*if-then-else*" : en effectuant le remplacement pour tous les nœuds des sous-arbres "*then*" et "*else*".

S'il y a eu des coupures, les destinations, attributs *In*, *Out* et les post-affectations doivent être recalculés pour tous les nœuds dont un sous-arbre contient une variable qui a été remplacée.

Exemple 4.9: (suite de l'exemple 4.8 page 99).



Les arcs orientés indiquent l'ordre d'évaluation entre nœuds à respecter, l'arc de $\mathcal{R}_<$ qui permet d'éviter les cycles est supprimé et la duplication de la variable en cause est introduite en attribut *sauvegardes* de $\mathcal{N}_1$ (trait gras continu). La simple utilisation d'une variable est notée par un arc formé d'un trait simple, une modification ou affectation par un trait double.

■

4.7 Traduction du graphe d'évaluation de T sans cycle en corps de programme impératif

De l'axiomatisation de l'opérateur défini c à dériver, nous avons obtenu un arbre abstrait qui code un terme T à dériver. Nous avons étendu l'arbre abstrait par des arcs qui codent les ordres d'évaluation à respecter entre les sous-termes de T, ce qui forme un graphe orienté sans cycle. Nous avons étendu le graphe par des arcs qui codent un ordre d'évaluation dans le domaine concret des termes représentés, qui, s'il est respecté évite les conflits d'accès aux variables. Ce graphe peut être cyclique. Nous avons supprimé ces cycles en ôtant certains des arcs que nous venions d'ajouter pour éviter des conflits, et réglé ces conflits en ajoutant des duplications de sauvegarde des variables redevenues conflictuelles (arcs choisis de façon à avoir un coût minimum en duplications).

Nous avons finalement obtenu un graphe orienté sans cycle où chaque nœud, qui représente un terme, s'il est origine d'un arc, doit et peut être évalué avant le nœud extrémité de cet arc (c'est-à-dire que son texte impératif dérivé est généré de manière à être exécuté avant celui du nœud extrémité), soit parce qu'il en est un sous-terme, soit pour régler un conflit.

La traduction de l'arbre abstrait en un texte Ada se fait alors par une interprétation de chaque nœud dans le domaine concret. Nous parcourons ce graphe tout ordre compatible avec l'ordre défini par les arcs du graphe (en profondeur ou en largeur d'abord, de gauche à droite ou inversement pour les opérandes, et les calculs anticipés avant, parmi, ou après les opérandes) et générons pour chaque nœud à évaluer son code correspondant en impératif.

La partie fonctionnelle d'un calcul anticipé doit être exécutée avec sa partie procédurale

La partie fonctionnelle d'un calcul anticipé ne peut être placée en paramètre effectif : elle n'est alors pas exécutée avant la partie procédurale des calculs qu'elle doit précéder.

Un nœud qui a été dérivé de manière à ce que le terme t' qu'il code soit calculé hors du calcul du terme t auquel il appartient (t' sous-terme de t), s'il voit sa partie fonctionnelle utilisée telle quelle comme paramètre effectif, peut ne plus respecter l'ordre d'évaluation qui a motivé son anticipation.

Exemple 4.10: Soit le terme $\mathrm{T} = append(append(x, y), reverse(x))$ à dériver et les implémentations suivantes des opérateurs qu'il comporte :

append est implémenté par une fonction :
$\mathcal{E}(append) = [x, y]\{\texttt{APPEND}(x, y)\}$,
$In(append) = < \pi_1, \pi_2 >$,
$Out(append) = < \phi, \phi >$.

reverse est implémenté par une procédure qui modifie son unique paramètre :
$\mathcal{E}(reverse) = [x]\{\texttt{REVERSE}(x)\}$,
$In(reverse) = < \pi_1 >$,
$Out(reverse) = < \pi_1 >$.

Nous notons $\mathcal{N}_1$ le nœud codant T et respectivement $\mathcal{N}_2$ et $\mathcal{N}_3$ pour les sous-termes $append(x, y)$ et $reverse(x)$.

Nous souhaitons dériver le nœud $\mathcal{N}_2$ avant $\mathcal{N}_3$ pour éviter que la valeur de x ne soit modifiée avant son utilisation par $\mathcal{N}_2$. La dérivation de $\mathcal{N}_2$ engendre un code fonctionnel : $([\![\,]\!], \langle\!|\texttt{APPEND}(x, y)|\!\rangle)$ et celle de $\mathcal{N}_3$ un code procédural : $([\![\texttt{REVERSE}(x)]\!], \langle\!|x|\!\rangle)$. En générant le code Ada par un parcours de gauche à droite, le nœud $\mathcal{N}_2$ est apparemment déjà dérivé pour $\mathcal{N}_3$, mais le code engendré non encore exécuté s'il est inclus dans la partie fonctionnelle de $\mathcal{N}_1$: $([\![\texttt{REVERSE}(x)]\!], \langle\!|\texttt{APPEND}(\texttt{APPEND}(x, y), x)|\!\rangle)$. Si T est le terme en partie droite de l'équation à dériver nous obtenons :

```
begin
  REVERSE(x);
  return APPEND(APPEND(x,y),x);
end;
```

qui est incorrect. Nous devons faire en sorte que tout le texte dérivé vers une forme fonctionnelle et qui doit être exécuté par anticipation soit évalué avant tout code procédural qu'il doit précéder. La façon correcte est que le code engendré pour $\mathcal{N}_2$ (partie fonctionnelle et partie procédurale) apparaisse avant le code engendré pour $\mathcal{N}_3$, c'est-à-dire :

```
begin
  v1 := APPEND(x,y);
  REVERSE(x);
  return APPEND(v1,x);
end;
```

■

Algorithme

Nous proposons de faire un parcours de l'arbre abstrait en profondeur d'abord, de gauche à droite pour les opérandes, et les nœuds *antecedents* (calculs anticipés déterminés par les arcs $\mathcal{R}_<$) après les opérandes, en générant le texte impératif de chaque nœud dans l'ordre de dépilement inverse (de la pile de retour en arrière pour le parcours de l'arbre). Nous partons du nœud racine $\mathcal{N}_1$,et chaque nœud traduit est annoté $execute = vrai$ de manière à ce que le terme calculé par anticipation ne soit pas re-calculé lorsque sa valeur est nécesaire.

La dérivation $\Delta^{\#}(\mathcal{N})$ se fait de la manière suivante :

- Cas "le code du nœud $\mathcal{N}$ a déjà été généré" ($\mathcal{N}.execute = vrai$) : le code en retour est $([\![]\!], \langle\!|f_1, \ldots, f_n|\!\rangle)$ pour $\mathcal{N}.destination = (f_1, \ldots, f_n)$.
- Cas "le code du nœud $\mathcal{N}$ n'a pas encore été généré" ($\mathcal{N}.execute = faux$). Nous obtenons ce code $([\![p]\!], \langle\!|f|\!\rangle)$ de la manière suivante :
  1. nous générons le code nécessaire aux sauvegardes créées lors de l'application des coupures de cycles :

$$\forall i \in \mathcal{N}.\, sauvegardes \cdot [\![p]\!] := [\![p]\!] + \Delta^{\#}(\mathcal{N}_i)$$

avec $\mathcal{N}_i$ qui code $duplication(x, y)$,

2. pour tous ses nœuds opérandes $\mathcal{N}'$ ($i \in \mathcal{N}.\, operandes \ \wedge \ \mathcal{N}' = \mathcal{N}_i$), nous générons le code correspondant :

$$\Delta^{\#}(\mathcal{N}') = (\llbracket p'_1, \ldots, p'_k \rrbracket, \langle\!| f'_1, \ldots, f'_k |\!\rangle)$$

et ajoutons la partie procédurale à celle déjà obtenue :

$$\forall i \in \mathcal{N}.\, operandes \cdot \llbracket p \rrbracket := \llbracket p \rrbracket + \llbracket p'_i \rrbracket)$$

3. pour tous ses nœuds $\mathcal{N}''$ dont le calcul doit être anticipé pour être exécuté avant $\mathcal{N}$ ($i \in \mathcal{N}.\, antecedents \cup \mathcal{N}.\, op\text{-}dependance \ \wedge \ \mathcal{N}'' = \mathcal{N}_i$), nous générons le code correspondant :
soit $\Delta^{\#}(\mathcal{N}'') = (\llbracket p''_1, \ldots, p''_k \rrbracket, \langle\!| f''_1, \ldots, f''_k |\!\rangle)$, avec $f''_1, \ldots, f''_k$ qui sont des variables, égales à la destination de $\mathcal{N}''$ (ce ne sont pas des expressions du langage concret, même et surtout si $\mathcal{N}''$ est également un nœud opérande de $\mathcal{N}$), et ajoutons la partie procédurale à celle déjà obtenue :

$$\forall i \in \mathcal{N}.\, antecedents \cup \mathcal{N}.\, op\text{-}dependance \cdot \llbracket p \rrbracket := \llbracket p \rrbracket + \llbracket p''_i \rrbracket)$$

4. nous générons le code implémentant l'application de l'opérateur du nœud sur ses opérandes :

   - pour un constructeur ou opérateur défini :
     selon la règle 3.4 page 68, avec les représentations des opérandes dans $\langle\!| f'_1, \ldots, f'_k |\!\rangle$. Nous ne dérivons les post-affectations que si $\mathcal{N}.\, anticipe = vrai$. Nous obtenons :
     $$(\llbracket p^{op} \rrbracket, \langle\!| f^{op} |\!\rangle),$$
     (lorsque $\mathcal{N}.\, anticipe = vrai$, $\langle\!| f^{op} |\!\rangle = \mathcal{N}.\, destination$, et ajoutons la partie procédurale à celle déjà obtenue :
     $$\forall i \in \mathcal{N}.\, antecedents \cup \mathcal{N}.\, op\text{-}dependance \cdot \llbracket p \rrbracket := \llbracket p \rrbracket + \llbracket p \rrbracket^{op})$$
     et $\langle\!| f |\!\rangle = \langle\!| f^{op} |\!\rangle$,
   - pour l'opérateur prédéfini *let* :
     nous avons en premier opérande la variable v définie, en second, le terme qui est affecté à cette variable, et en troisième le terme dont la valeur est la valeur du *let*, et les dérivations correspondantes : $(\llbracket p_1 \rrbracket, \langle\!| f_1 |\!\rangle)$, $(\llbracket p_2 \rrbracket, \langle\!| f_2 |\!\rangle)$ et $(\llbracket p_3 \rrbracket, \langle\!| f_3 |\!\rangle)$, avec $\llbracket p_1 \rrbracket = \llbracket\ \rrbracket$, $\langle\!| f_1 |\!\rangle = \langle\!| v |\!\rangle$ et v de sorte s_v. Nous obtenons :

```
( ⟦ declare
      v : Δ_S(s_v)
    begin
      p_2;
      v := f_2;
      p_3;
    end ; ⟧
  ,⟨|f_3|⟩ )
```

     Nous ajoutons la partie procédurale à celle déjà obtenue, et $\langle\!| f |\!\rangle = \langle\!| f_3 |\!\rangle$,

– pour l'opérateur prédéfini *if-then-else*, avec $\mathcal{N}.operandes = (id_1, id_2, id_3)$, nous avons :
$\Delta^{\#}(\mathcal{N}_{id_1}) = (\llbracket p_1 \rrbracket, \langle f_1 \rangle)$,
$\Delta^{\#}(\mathcal{N}_{id_2}) = (\llbracket p_{2,1}, \ldots, p_{2,n} \rrbracket, \langle f_{2,1}, \ldots, f_{2,n} \rangle)$,
$\Delta^{\#}(\mathcal{N}_{id_3}) = (\llbracket p_{3,1}, \ldots, p_{3,n} \rrbracket, \langle f_{3,1}, \ldots, f_{3,n} \rangle)$,
avec $\forall i \in [1, \ldots, n] \cdot (f_{2,i} = f_{3,i})$.
Nous obtenons :

```
( [[p1; if   f1  then  begin
                         p2,1; ... ; p2,n;
                       end ;
       else
                       begin
                         p3,1; ... ; p3,n;
                       end ;
       endif ;]]
  ,<f2,1, ..., f2,n> )
```

Nous ajoutons la partie procédurale à celle déjà obtenue, et $\langle f \rangle = \langle f_{2,1}, \ldots, f_{2,n} \rangle$,

5. $\mathcal{N}.execute := vrai$.

Après avoir parcouru tout le graphe, nous avons donc :

- $\Delta^{\#}{}_{(x_1,\ldots,x_m)}(\mathcal{N}_1)$ qui donne $(\llbracket p \rrbracket, \langle x_1, \ldots, x_m \rangle)$, avec $m \geq 1$, la partie fonctionnelle étant vide pour une dérivation d'un opérateur *op* : $(s_1, \ldots, s_n)$ -> $(s_{n+1}, \ldots, s_{n+m})$ en une procédure et en ayant décoré le nœud $\mathcal{N}_1$ avec ses destinations calculées : $\mathcal{N}_1.destination = (x_1, \ldots, x_n)$.
- $\Delta^{\#}(\mathcal{N}_1)$, cas de $\Delta^{\#}{}_{(x_1,\ldots,x_m)}$ avec $m = 0$ qui donne $(\llbracket p \rrbracket, \langle f \rangle)$ où p est éventuellement vide, pour une dérivation vers une fonction. Le corps du programme est alors "`p; return f;`".

Exemple 4.11: (suite de l'exemple 4.9 page 101).
Un parcours en profondeur d'abord, les nœuds fils, puis les nœuds opérandes, en montrant les nœuds sur la pile de retour-en-arrière est :

$\mathcal{N}_8$:
$(\llbracket p_8 \rrbracket, \langle f_8 \rangle) = (\llbracket duplication(z, z_2); \rrbracket, \langle z_2 \rangle)$
$\mathcal{N}_1 \leftarrow \mathcal{N}_5 \leftarrow \mathcal{N}_6 \leftarrow \mathcal{N}_7$
$(\llbracket p_7 \rrbracket, \langle f_7 \rangle) = (\llbracket p_8;$
$renverse(z_2); \rrbracket, \langle z_2 \rangle)$
$(\llbracket p_6 \rrbracket, \langle f_6 \rangle) = (\llbracket p_7;$
$v_1 := concat_3(x, y, z_2); \rrbracket, \langle v_1 \rangle)$
$(\llbracket p_5 \rrbracket, \langle f_5 \rangle) = (\llbracket p_6;$
$concat(v_1, t); \rrbracket, \langle v_1 \rangle)$
$\mathcal{N}_4 \leftarrow \not{\mathcal{N}_6} \leftarrow \mathcal{N}_3 \leftarrow \not{\mathcal{N}_5}$
$(\llbracket p_3 \rrbracket, \langle f_3 \rangle) = (\llbracket p_5;$
$concat(t, x); \rrbracket, \langle t \rangle)$

$$(\llbracket p_4 \rrbracket, \langle\!| f_4 |\!\rangle) = (\llbracket p_3 ; \\ concat(x, z); \rrbracket, \langle\!| x |\!\rangle)$$

$\mathcal{N}_2$:

$$(\llbracket p_2 \rrbracket, \langle\!| f_2 |\!\rangle) = (\llbracket p_4 ; \\ v_2 := tri(t); \rrbracket, \langle\!| v_2 |\!\rangle)$$

$$(\llbracket p_2 ; \\ duplication(v_1, z); \\ duplication(v_2, t); \rrbracket, \langle\!| z, x, t |\!\rangle)$$

Nous obtenons donc le texte suivant :

$$\begin{array}{l} duplication(z, z_2); \\ renverse(z_2); \\ v_1 := concat_3(x, y, z_2); \\ concat(v_1, t); \\ concat(t, x); \\ concat(x, z); \\ v_2 := tri(t); \\ duplication(v_1, z); \\ duplication(v_2, t); \end{array}$$

auquel nous devons ajouter la déclaration des variables locales (variables "fraîches" de la dérivation) pour former le corps de la procédure :

```
procedure OP(x : in out listint; y : in listint; z, t : in out listint)
```

■

Conclusion

Dans ce chapitre nous avons proposé une méthode de résolution des conflits d'accès aux variables qui apparaissent lorsque l'on effectue, dans le domaine concret, une simulation des calculs nécessaires à l'obtention, dans le domaine abstrait, de la valeur d'un terme T.

Nous avons d'abord présenté comment localiser les conflits dans le terme T à dériver :

- entre sous-termes de T : $op(t_1, \ldots, t_n)$ et $op'(t'_1, \ldots, t'_n)$, dont les textes engendrés pour simuler l'application de op et op' sur respectivement $(t_1, \ldots, t_n)$ et $(t'_1, \ldots, t'_n)$ ont des accès à une (ou des) variable(s) selon des modes différents (lecture par rapport à écriture ou lecture-écriture) ;

- entre les affectations qui sont rajoutées lorsque des résultats de calculs doivent être contenus dans des variables précises (paramètres formels de retour, variables identiques pour les deux branches alternatives d'un *if-then-else*).

Chaque conflit peut être résolu, soit par un déplacement de code généré de manière à imposer l'ordre dans lequel sont effectués certains calculs, soit par une duplication de sauvegarde de la valeur de la variable conflictuelle.

Les ordres dans lesquels doivent être effectués les calculs, pour résoudre l'ensemble des conflits, peuvent être contradictoires et introduire des cycles. Nous avons donc proposé une méthode de résolution de l'ensemble des conflits, qui minimise le coût en duplication de variables.

Nous avons ensuite montré comment obtenir un texte à la syntaxe Ada, à partir du graphe constitué de l'arbre abstrait et des ordres d'exécution des calculs, auxquels nous avons ajouté un ordre de parcours de l'arbre compatible.

Le texte Ada est correct par construction : nous supposons que les implémentations existantes des opérateurs sont correctes, et notre algorithme est basé sur la conservation d'une valeur correcte des représentation des opérandes.

Soit $t = op(t_1, \ldots, t_n)$ le terme à dériver, notre hypothèse est que la dérivation de $t_1, \ldots, t_n$ est correcte, c'est-à-dire que nous obtenons :

$$(\llbracket instructions_{id_1} \rrbracket, \langle\!| expression_{id_1} |\!\rangle), \ldots, (\llbracket instructions_{id_n} \rrbracket, \langle\!| expression_{id_n} |\!\rangle)$$

avec $\{id_1, \ldots, id_n\} = \{1, \ldots, n\}$ selon l'ordre choisi pour le parcours de l'arbre abstrait (en largeur ou en profondeur d'abord, de gauche à droite selon un ordre quelconque parmi les opérandes). Chaque expression fonctionnelle $\langle\!| expression_{id_{i \in [1 \ldots n]}} |\!\rangle$ est évaluée après l'exécution de $\llbracket instructions_{id_1} ; \ldots ; instructions_{id_i} \rrbracket$ avec pour valeur celle de t_{id_i}.

L'implémentation de *op* étant correcte par hypothèse, la correction du résultat de l'application $\Delta(op)$ sur $(expression_1, \ldots, expression_n)$ après avoir effectué les duplications de sauvegarde, puis exécuté $\llbracket instructions_{id_1} ; \ldots ; instructions_{id_n} \rrbracket$, est conditionnée par la non destruction de toute variable de $expression_{id_i}$ par $\llbracket instructions_{id_{i+1}} ; \ldots ; instructions_{id_n} \rrbracket$. La preuve de cette correction est triviale : toute modification d'une variable x de $expression_{id_i}$ (qui est un paramètre formel ou une variable intermédiaire définie par un "*let*") par une instruction de $instructions_{id_{j \in [i+1 \ldots n]}}$ ne peut exister. Sinon, un ordre de précédence entre les calculs de t_{id_i} et de t_{id_j} existerait et imposerait la préservation de x ou sa duplication.

Chapitre 5

Compilation des constructeurs

5.1 Dérivation systématique

Une dérivation d'un type algébrique dans le domaine concret peut être faite de manière systématique en dérivant le type LPG en un paquetage Ada dont le nom et ceux des fichiers (interfaces et corps) correspondants sont donnés par le lien d'implémentation (ceci est décrit dans le section 7.2 page 131).

Nous nous limitons aux spécifications algébriques paramétrées au maximum d'une sorte formelle. Les sortes connues sont les sortes de la bibliothèque pour lesquelles nous avons choisi une implémentation (via son lien). Ainsi, une déclaration de type :

```
Type S requires Formal_Sort[s']
sorts
    s
constructors
⋮
end
```

est dérivée en un paquetage dont la spécification est :

```
generic
    type t_s' is private ;
package p_S is
  type t_s is ...
⋮
end p_S ;
```

et le lien d'implémentation $\Delta_S(s', \texttt{t\_}s')$ est ajouté pour la dérivation.

Nous dérivons l'interface Ada des constructeurs et opérateurs tel que cela est décrit en section 3.2.1 page 60. La dérivation automatique du corps d'une fonction ou procédure qui implémente un opérateur défini, à partir de son axiomatisation, est détaillée au chapitre précédent. Nous pouvons dériver automatiquement les constructeurs de la sorte s de la manière suivante :

Soit une sorte s disposant de p constructeurs tels que :

$$c_{i\in[1\ldots p]} : (s_{i,1}, \ldots, s_{i,d_i}) \text{ -> } s$$

avec $\forall i \in [1 \dots p] \cdot (d_i = Card(Dom(c_i)))$ la cardinalité du domaine de $c_{i \in [1 \dots p]}$.
Soit `t_`$s = \Delta_S(s)$ et les types qui implémentent les sortes des domaines des constructeurs de s :

$$\forall i \in [1 \dots p], j \in [1 \dots d_i] \cdot t_{i,j} = \Delta_S(s_{i,j})$$

Le type `t_`s peut être défini de la manière suivante :

```
type constructeur_de_t_s is (c_1, ..., c_p) ;
type t_s is access block_t_s ;

type block_t_s(cstr : constructeur_de_t_s) is record
  case cstr is
    when c_1 => c_1_1 : t_{1,1}
                ⋮
                c_1_d_1 : t_{1,d_1}
    ⋮
    when c_p => c_p_1 : t_{p,1}
                ⋮
                c_p_d_p : t_{p,d_p}
  end case ;
end record ;
```

Pour la sorte s qui possède p constructeurs, définie par le type algébrique S, le lien d'implémentation de S par le paquetage Ada `p_S` est composé de :

$\Delta_S(s) =$ `t_`s

$\forall i \in [1 \dots p] \cdot \Delta_C(c_i) = (\mathcal{E}(c_i), In(c_i), Out(c_i), Is(c_i), Select(c_i))$

qui sont définis par :

$$\forall i \in [1 \dots p], n = d_i \cdot \begin{cases} \mathcal{E}(c_i) = [v_1, \dots, v_n]\{\texttt{new t\_s'(cstr =>}c_i\texttt{;} \\ \qquad\qquad\qquad\qquad\qquad\qquad c_i\_1\texttt{=>}v_1\texttt{;} \\ \qquad\qquad\qquad\qquad\qquad\qquad \vdots \\ \qquad\qquad\qquad\qquad\qquad\qquad c_i\_n\texttt{=>}v_n\texttt{)}\} \\ In(c_i) = < \pi_1, \dots, \pi_n > \\ Out(c_i) = < \phi, \dots, \phi > \end{cases}$$

Le testeur is^c du domaine concret (Ada) permet de tester si une expression de type t (t implémentant la sorte s dont c est l'un des constructeurs) est la représentation d'un terme algébrique construit par c :

$\mathcal{E}(is^c) = [v]\{$`v.cstr = c`$\}$,
$In(is^c) = < \pi_1 >$,
$Out(is^c) = < \phi >$

Pour chaque constructeur, le nombre de sélecteurs dont nous avons besoin est égal à la cardinalité du domaine du constructeur. Chaque sélecteur du domaine abstrait est défini de la manière suivante pour un constructeur $c : (s_1, \ldots, s_n) \rightarrow s$:

$$\forall i \in [1 \ldots n],\ \forall v_1 : s_1,\ \ldots, \forall v_n : s_n\ \cdot select_i^c(c(v_1, \ldots, v_n)) = v_i$$

Il s'agit d'écrire l'implémentation des sélecteurs qui correspondent aux constructeurs dérivés automatiquement :

$$\forall i \in [1 \ldots p], j \in [1 \ldots d_i] \cdot \left(\begin{array}{l} \mathcal{E}(select_j^{c_i}) = [v]\{v.c_i\_j\} \\ In(select_j^{c_i}) = < \pi_1 > \\ Out(select_j^{c_i}) = < \phi > \end{array} \right)$$

De plus, nous avons $LeftExpr(select_j^{c_i}) = vrai$ (ce qui n'est pas utilisé dans la dérivation définie au chapitre prédent, où seules les variables sont des expressions-gauches).

Nous définissons également pour toutes les implémentations de chaque sorte, deux fonctions Ada : `eq` et `duplication`. La fonction booléenne `eq` donne l'égalité entre deux valeurs, qui peuvent être deux représentations différentes d'une même valeur algébrique (dans le cas d'insertions d'éléments dans un ensemble, par exemple, le résultat de l'exécution du texte Ada engendré pour $ins(x, ins(y, l))$ peut être différent de celui obtenu pour $ins(y, ins(x, l))$). La fonction `duplication`, à partir d'une représentation d'une valeur algébrique, construit une nouvelle valeur qui ne partage aucun emplacement en mémoire avec la précédente.

```
function eq(x,y : t_s) return boolean is
begin
  if not eq(x.cstr,y.cstr) then return false
  else
    case x.cstr is
      when c_1 => return eq(x.c_1_1, y.c_1_1)
                  and
                  ⋮
                  and eq(x.c_1_d_1, y.c_1_d_1);
      ⋮
      when c_p => return eq(x.c_p_1, y.c_p_1)
                  and
                  ⋮
                  and eq(x.c_p_d_p, y.c_p_d_p);
    end case ;
end eq ;
```

```
function duplication(x : t_s) return t_s is
begin
  case x.cstr is
    when c_1 => return new t_s'(E(c_1)(duplication(x.c_1_1),
                                       ⋮
                                       ,duplication(x.c_1_d_1))
    ⋮
    when c_p => return new t_s'(E(c_p)(duplication(x.c_p_1),
                                       ⋮
                                       ,duplication(x.c_p_d_p))
  end case ;
end eq ;
```

5.2 Implémentation par un type existant

5.2.1 La bibliothèque

Les types algébriques *Nat, Char, String, Array, Seq* et *Set* prédéfinis en LPG peuvent être implémentées par des paquetages prédéfinis en Ada.

Exemple 5.1: Soit le type des séquences génériques :

```
Type Seq requires Formal_Sort[elem]
sorts
    seq[elem]
constructors
    nil : -> seq[elem]
    cons : (elem, seq[elem]) -> seq[elem]
operators
    is_nil : seq[elem] -> bool
    car : seq[elem] -> elem
    cdr : seq[elem] -> seq[elem]
equations
⋮
end Seq
```

Il peut être implémenté en Ada, pour $\Delta_S(elem) = elem$, par le paquetage classique des listes construites par un pointeur soit égal à `null`, soit qui pointe sur un bloc contenant une valeur et un pointeur sur la suite de la liste (il n'y a pas de partie variante, comme pour la dérivation automatique) :

$\Delta_S(seq) = list$

$\Delta_C(nil) = (\mathcal{E}(nil) = [\,]\{\texttt{null}\}, In(nil) = <>, Out(nil) = <>)$

$\Delta_C(cons) = (\ \mathcal{E}(cons) = [x, l]\{\texttt{new list'(val =>x; suiv =>l)}\},$
$In(cons) = < \pi_1, \pi_2 >,$
$Out(cons) = < \phi, \phi >)$

$\Delta_D(is^{nil}) = (\mathcal{E}(is^{nil}) = [x]\{\texttt{(x=null)}\},\ In(is^{nil}) = <\pi_1>,\ Out(is^{nil}) = <\phi>)$

$\Delta_D(is^{cons}) = (\mathcal{E}(is^{cons}) = [x]\{\texttt{(x /= null)}\},\ In(is^{cons}) = <\pi_1>,\ Out(is^{cons}) = <\phi>)$

$$\begin{array}{l}\Delta_D(select_1^{cons}) = (\ \mathcal{E}(select_1^{cons}) = [x]\{\texttt{x.val}\},\\ \qquad\qquad\qquad\qquad In(select_1^{cons}) = <\pi_1>,\\ \qquad\qquad\qquad\qquad Out(select_1^{cons}) = <\phi>)\\ \quad LeftExpr(select_1^{cons}) = vrai\end{array}$$

$$\begin{array}{l}\Delta_D(select_2^{cons}) = (\ \mathcal{E}(select_2^{cons}) = [x]\{\texttt{x.suiv}\},\\ \qquad\qquad\qquad\qquad In(select_2^{cons}) = <\pi_1>,\\ \qquad\qquad\qquad\qquad Out(select_2^{cons}) = <\phi>)\\ \quad LeftExpr(select_2^{cons}) = vrai\end{array}$$

Nous ajoutons les fonctions Ada `eq` et `duplication` qui implémentent respectivement l'égalité entre termes d'une même sorte, et la construction d'un terme identique à celui passé en opérande. Ces deux fonctions doivent être fournies pour toute implémentation en Ada d'un type algébrique.

$\Delta(eq) = (\mathcal{E}(eq) = [x,y]\{\texttt{eq(}x\texttt{,}y\texttt{)}\},\ In(eq) = <\pi_1,\pi_2>,\ Out(eq) = <\phi,\phi>)$

$$\begin{array}{l}\Delta(duplication) = (\ \mathcal{E}(duplication) = [x,y]\{y\ \texttt{:= duplication(}x\texttt{);}\},\\ \qquad\qquad\qquad\qquad In(duplication) = <\pi_1,\phi>,\\ \qquad\qquad\qquad\qquad Out(duplication) = <\phi,\pi_2>)\end{array}$$

■

5.2.2 Optimisation : réutilisation de l'implémentation des constructeurs

L'optimisation de la dérivation des constructeurs consiste à utiliser des représentations de données qui sont plus efficaces que celles de la dérivation systématique, en termes de taille mémoire ou de rapidité d'accès. Les critères à respecter sont :

- la réalisabilité : tout terme du domaine abstrait possède (au moins) une représentation dans le domaine concret,
- la consistance : un terme du domaine concret ne peut représenter deux termes différents du domaine abstrait,
- l'implémentation de chaque constructeur doit être correcte, c'est-à-dire que la règle 3.3 d'interprétation d'un terme abstrait dans le domaine concret définie page 61 est respectée.

La plupart des optimisations classiques se généralisent par la réutilisation d'une implémentation existante : en effet, les critères qui conduisent à implémenter une sorte par le type entier, le type octet, le type liste, le type arbre binaire, ..., peuvent être définis par l'existence d'un lien entre les constructeurs de la spécification à dériver, et ceux de la spécification LPG du type entier, octet, liste, arbre binaire, etc.

Réutiliser pour une sorte s', l'implémentation d'une sorte s, implique que pour chaque constructeur ou opérateur défini de s' il existe un constructeur ou opérateur défini de s qui ait la même sémantique. Si nous pouvons définir et vérifier une équivalence entre deux signatures algébriques par l'existence d'un isomorphisme entre elles, vérifier une équivalence entre deux spécifications où des opérateurs sont définis par des axiomes est extrêmement complexe. Cela n'est pas décidable dans le cas général (équivalence de deux théories par preuve de l'équivalence de deux axiomatisations, ou équivalence de deux constructions modulaires [Ori96]) et n'est pas notre propos.

Par contre, nous pouvons définir si l'ensemble des constructeurs de s' est équivalent à un sous-ensemble des constructeurs de s, et peut réutiliser l'implémentation de ces derniers (et gagner ainsi en efficacité par rapport à une dérivation systématique des constructeurs).

Cette approche est décrite dans [Sig95], qui est une continuation du travail effectué dans [Tur92]. Nous allons généraliser ce travail pour permettre d'implémenter un constructeur par un autre qui ne lui est pas isomorphe (car les arités sont différentes), mais dont le domaine de l'un est une permutation du domaine de l'autre.

Dans les définitions suivantes, nous appellons bijection une fonction totale bijective, et injection une fonction totale injective.

Définition 5.1: Un combinateur In (ou Out) bijectif est un combinateur In (ou Out) $< P_1, \ldots, P_n >$ tel que :

$$\begin{cases} \forall i \in [1 \ldots n] \cdot P_i \neq \phi \\ \forall i \in [1 \ldots n] \cdot \pi_1 \in \{P_1, \ldots, P_n\} \end{cases}$$

Plus simplement, $< P_1, \ldots, P_n >$ est une permutation de $< \pi_1, \ldots, \pi_n >$.

■

Propriété :

Soit B un combinateur In (ou Out) bijectif, alors $B \circ B^{-1} = B^{-1} \circ B = id_{()_n}$. Le combinateur $id_{()_n}$ est le combinateur identité pour les produits cartésiens de n éléments. Il est égal à : $< \pi_1, \pi_2, \ldots, \pi_n >$

Définition 5.2: Nous appelons $\Delta^I_{SP'} : \Sigma' \rightarrow \Sigma$ lien d'implémentation abstraite, une application de la signature $\Sigma' = (S', \Omega')$ d'une spécification algébrique SP' vers la signature $\Sigma = (S, \Omega)$ de la spécification algébrique SP.

- $\Delta^I_{S'}$ est une application qui, à une sorte $s' \in S'$ associe une sorte $s \in S$,
- $\Delta^I_{C'}$ est composé de $(\mathcal{E}^I_{C'}, In^I_{C'}, Out^I_{C'})$:

  Nous omettons de préciser l'ensemble de constructeurs domaine des composants de $\Delta^I_{C'}$ lorsqu'il n'y a pas d'ambiguïté.

  $\mathcal{E}^I$ une injection de l'ensemble $C'_{s'}$ des constructeurs de $s' \in S'$ vers l'ensemble C_s des constructeurs de $s = \Delta^I_S(s')$.

  In^I est une fonction qui associe à un constructeur $c' \in C'_{s'}$ un combinateur In bijectif,

Out^I est une fonction qui associe à un constructeur $c' \in C'_{s'}$ un combinateur Out bijectif.

qui sont tels que, pour :

$\Delta^I_S(s') = s$
$c' : (s'_1, \ldots, s'_n) \text{ -> } s'$ un constructeur appartenant à $C'_{s'}$
$\mathcal{E}^I(c') = c \in C_s$

nous avons :

$$c : (s_1, \ldots, s_n) \text{ -> } s$$
$$In^I(c')(\Delta^I_{S'}(s'_1), \ldots, \Delta^I_S(s'_n)) = (s_1, \ldots, s_n)$$
$$Out^I(c')(\Delta^I_{S'}(s')) = s$$

$\mathcal{E}^I$ associe à un constructeur $c' \in C'_{s'}$ de s', le constructeur $c \in C_s$ qui l'implémente. $In^I(c')$ donne la correspondance entre les opérandes de $c' \in C'_{s'}$ et ceux de $c = \mathcal{E}^I(c') \in C_s$. $Out^I(c')$ donne la correspondance entre les résultats de l'application de $c' \in C'_{s'}$ sur ses opérandes $t_1 \ldots t_n$, et les résultats de l'application de $c = \mathcal{E}^I(c') \in C_s$ sur $In^I(c')(t_1, \ldots, t_n)$. $Out^I(c') = < \pi_1 >$, quel que soit le constructeur c, les constructeurs ayant une seule sorte dans leur co-domaine.

■

Propriétés : l'algèbre des termes $\mathrm{T}_{\Sigma'=(s',C'_{s'})}$ n'est pas isomorphe à $\mathrm{T}_{\Sigma''=(s,\Delta^I_{C'}(C'_{s'}))}$, car les arités des constructeurs $C'_{s'}$ ne sont pas les mêmes que celles des constructeurs qui les implémentent, mais il existe un isomorphisme d'ensembles (une bijection) entre l'ensemble engendré par les constructeurs de $C'_{s'}$ et celui engendré par les constructeurs de $\mathcal{E}^I_{C'}(C'_{s'})$ que l'on peut décrire à partir de Δ^I.

Soit $c' : (s'_1, \ldots, s'_n) \text{ -> } s'$ un constructeur de $C'_{s'}$ et $c : (s_1, \ldots, s_n) \text{ -> } s$ de C_s qui l'implémente, avec $\mathcal{E}^I_{C'}(c') = c$ et $\Delta^I_{S'}(s') = s$.
Soit $p_c : In^I_{C'}(c')^{-1}(\Delta^I_{S'}(s'_1), \ldots, \Delta^I_{S'}(s'_n)) \text{ -> } s$ un opérateur défini par l'unique équation :

$$p_c(x_1, \ldots, x_n) \Longrightarrow c(In^I_{C'}(c')^{-1}(x_1, \ldots, x_n))$$

et $\Omega^{c'}_c$ l'ensemble des opérateurs p_c ainsi définis.
Alors les algèbres $\mathrm{T}_{\Sigma'=(s',C'_{s'})}$ et $\mathrm{T}_{\Sigma'''=(s,C_s \cup \Omega^{c'}_c)}\Big|_{s,\Omega^{c'}_c}$ sont isomorphes.

Réutilisation par composition avec un lien d'implémentation abstraite
Le principe de la réutilisation de l'implémentation des constructeurs de s pour implémenter ceux de s' (et de pouvoir ainsi dériver l'axiomatisation des opérateurs de s' en fonction des testeurs et sélecteurs de s) est d'effectuer une composition du lien d'implémentation défini au chapitre 3 et du lien d'implémentation abstraite que nous venons de définir.

Nous limitons ce lien d'implémentation abstraite aux sortes et aux constructeurs. Il peut être défini de manière similaire pour les opérateurs définis, mais nous ne pouvons

dans ce cas pas établir automatiquement l'équivalence entre deux axiomatisations (il est envisageable, si cette équivalence est démontrée, de réutiliser l'implémentation d'un opérateur défini pour implémenter un autre opérateur défini, dont nous ne dérivons alors pas l'axiomatisation).

Par contre, pour SP' la spécification à laquelle appartient s', il est possible de chercher automatiquement si la bibliothèque possède une spécification SP avec une sorte s telle qu'un lien d'implémentation abstraite $\Delta^I_{SP'}$ existe de la sorte s' et de ses constructeurs vers s et ses constructeurs. Cette recherche est faite de manière exhaustive pour chaque sorte et chaque sous-ensemble de ses constructeurs (ou par un algorithme semblable à celui écrit pour la recherche des coupures).

À partir du lien d'implémentation abstraite $\Delta^I_{SP'}$ trouvé, et d'un lien d'implémentation Δ_{SP}, nous pouvons calculer un lien d'implémentation $\Delta_{SP'} = \Delta_{SP} \circ \Delta^I_{SP'}$ pour s' et ses constructeurs :
$\Delta_{SP'}$ est défini par :

$\Delta_{S'} = \Delta_S \circ \Delta^I_{S'}$

$\forall c' \in C'_{s'} \cdot \Delta_{C'}(c') = (\mathcal{E}_{C'}(c'), In_{C'}(c'), Out_{C'}(c'), Is_{C'}(c'), Select_{C'}(c'))$ tels que :

$\mathcal{E}_{C'}(c') = \mathcal{E}_C(\mathcal{E}^I_{C'}(c'))$,
$In_{C'}(c') = In_C(\mathcal{E}^I_{C'}(c')) \circ In^I_{C'}(c')$,
$Out_{C'}(c') = Out_C(\mathcal{E}^I_{C'}(c')) \circ Out^I_{C'}(c')$,
$Is_{C'}(c') = is^{c'} = is^{\mathcal{E}^I_{C'}(c')}$
$\forall i \in [1 \ldots Card(Dom(c'))] \cdot select^{c'}_i =$
$< \pi_i > \circ (In^I_{C'}(c'))^{-1} \left(select_1^{\mathcal{E}^I_{C'}(c')}, \ldots, select^{\mathcal{E}^I_{C'}(c')}_{Card(Dom(\mathcal{E}^I_{C'}(c')))} \right)$

Exemple 5.2:
Soit la sorte s' suivante :

```
Type S' requires Formal_Sort[s'']
sorts
    s'
constructors
    c'1 : (s', s'') -> s'
    c'2 : -> s'
end
```

Alors il existe le lien d'implémentation Δ^I suivant entre la sorte s' et la sorte seq :

- $\Delta^I_{S'}(s') = s$ et $\Delta^I_{S'}(s'') = elem$
- $\Delta^I_{C'}(c'_1) = (\ \mathcal{E}^I_{C'}(c'_1) = \{\texttt{cons}\}, In^I_{C'}(c'_1) = < \pi_2, \pi_1 >, Out^I_{C'}(c'_1) = < \pi_1 >)$
- $\Delta^I_{C'}(c'_2) = (\ \mathcal{E}^I_{C'}(c'_2) = \{\texttt{null}\}, In^I_{C'}(c'_2) = <>, Out_{C'}(c'_2) = < \pi_1 >)$

La composition de l'implémentation abstraite et de l'implémentation concrète fait que la sorte s' est correctement implémentée en Ada par le type `list` vu précédemment (et le type algébrique S' par le paquetage $Liste$) avec le lien d'implémentation suivant :

- $\Delta_{S'}(s') = \Delta_S(\Delta^I_{S'}(s')) =$ `list`
- $\Delta_{C'}(c'_1) = (\mathcal{E}_{C'}(c'_1), In_{C'}(c'_1), Out_{C'}(c'_1), Is_{C'}(c'_1), Select_{C'}(c'_1))$

tel que :

$$
\begin{aligned}
\mathcal{E}_{C'}(c'_1) &= \mathcal{E}_C(\mathcal{E}^I_{C'}(c'_1)) \\
&= \mathcal{E}_C(cons) \\
&= [x,l]\{\texttt{new list'(val =>x; suiv =>l)}\} \\
In_{C'}(c'_1) &= In_C(\mathcal{E}^I_{C'}(c'_1)) \circ In^I_{C'}(c'_1) \\
&= In_C(cons) \circ In^I_{C'}(c'_1) \\
&= <\pi_1,\pi_2> \circ <\pi_2,\pi_1> \\
&= <\pi_2,\pi_1> \\
Out_{C'}(c'_1) &= Out_C(\mathcal{E}^I_{C'}(c'_1)) \circ Out^I_{C'}(c'_1) \\
&= Out_C(cons) \circ Out^I_{C'}(c'_1) \\
&= <\phi,\phi> \circ <\pi_1> \\
&= <\phi,\phi> \\
is^{c'_1} &= is^{\mathcal{E}^I_{C'}(c'_1)} \\
&= is^{cons} \\
is^{c'_2} &= is^{\Delta^I_{C'}(c'_2)} \\
&= is^{nil} \\
select_1^{c'_1} &= <\pi_1> \circ (In^I_{C'}(c'))^{-1} \left(select_1^{\mathcal{E}^I_{C'}(c'_1)}, \ldots, select^{\mathcal{E}^I_{C'}(c'_1)}_{Card(Dom(\mathcal{E}^I_{C'}(c')))} \right) \\
&= <\pi_1> \circ <\pi_2,\pi_1> (select_1^{cons}, \ldots, select_2^{cons}) \\
&= select_2^{cons} \\
select_2^{c'_1} &= <\pi_2> \circ (In^I_{C'}(c'))^{-1} \left(select_1^{\mathcal{E}^I_{C'}(c'_1)}, \ldots, select^{\mathcal{E}^I_{C'}(c'_1)}_{Card(Dom(\mathcal{E}^I_{C'}(c')))} \right) \\
&= <\pi_2> \circ <\pi_2,\pi_1> (select_1^{cons}, \ldots, select_2^{cons}) \\
&= select_1^{cons}
\end{aligned}
$$

■

5.2.3 Perspective : réutilisations partielles

Il s'agit de réutiliser pour implémenter un constructeur $c' : (s'_1, \ldots, s'_n) \rightarrow s'$, l'implémentation d'un constructeur $c : (s_1, \ldots, s_m) \rightarrow s$ avec $m \geq n$.

Définition 5.3: Nous appelons Δ^{I^+} le lien d'implémentation abstraite étendu, une généralisation de Δ^I :

- $\Delta^{I^+}_{S'}$ est une application qui, à une sorte $s' \in S'$ associe une sorte s qui l'implémente,

- $\Delta^{I^+}_{C'}$ est composé de $(\mathcal{E}^{I^+}_{C'}, In^{I^+}_{C'}, Out^{I^+}_{C'})$:

  $\mathcal{E}^{I^+}$ une injection de l'ensemble $C'_{s'}$ des constructeurs de $s' \in S$ vers l'ensemble C_s des constructeurs de $\Delta^{I^+}_{S'}(s') = s$.

  $In^{I^+}_{C'}$ est une fonction qui associe à un constructeur $c' \in C'_{s'}$ un combinateur In (pas obligatoirement bijectif),

  $Out^{I^+}_{C'}$ est une fonction qui associe à un constructeur $c' \in C'_{s'}$ un combinateur Out,

  qui sont tels que, pour :

  $\Delta^{I^+}_{S'}(s') = s$
  $c' : (s'_1, \ldots, s'_n) -> s'$ un constructeur appartenant à $C'_{s'}$
  $\mathcal{E}^{I^+}_{C'}(c') = c \in C_s$

  nous avons :

  $$c : (s_1, \ldots, s_m) -> s$$
  $$(In^{I^+}_{C'})^{-1}(c')(s_1, \ldots, s_n) = (\Delta^{I^+}_{S'}(s'_1), \ldots, \Delta^{I^+}_{S'}(s'_n))$$
  $$Out^{I^+}_{C'}(c')(\Delta^{I^+}_{S'}(s')) = s$$

  $\mathcal{E}^{I^+}_{C'}$ associe à un constructeur $c' \in C'_{s'}$ de s', le constructeur $c \in C_s$ qui l'implémente. $In^{I^+}_{C'}$ donne la correspondance entre les opérandes de $c' \in C'_{s'}$ et ceux de $\mathcal{E}^{I^+}_{C'}(c') = c \in C_s$. $Out^{I^+}_{C'}$ donne la correspondance entre les résultats de l'application de $c' \in C'_{s'}$ sur ses opérandes $t_1 \ldots t_n$, et les résultats de l'application de $\mathcal{E}^{I^+}(c') = c \in C_s$ sur $In^{I^+}_{C'}(c')(t_1, \ldots, t_n)$. $Out^{I^+}_{C'}(c') = < \pi_1 >$, quel que soit le constructeur c', les constructeurs ayant une seule sorte dans leur co-domaine.

■

Les combinateurs In^{I^+} ne sont plus obligatoirement des combinateurs bijectifs : ils ne sont plus composés uniquement de combinateurs de projection π_i et du combinateur ϕ, mais peuvent comporter une constante à la position de l'opérande de $c = \Delta^{I^+}_{C'}(c')$ qui est "inutile" pour implémenter c'. Par contre, ils doivent toujours respecter les autres caractéristiques de la définition 3.3 page 55.

Dès lors que nous savons construire une constante Ada pour tout type qui implémente une sorte, nous pouvons utiliser le lien d'implémentation $\Delta_C \circ \Delta^{I^+}_{C'}$ tel qu'il est défini dans la section 5.2.2 page 113.

La recherche du lien d'implémentation abstraite étendu peut être faite également de manière automatique.

Nous pouvons par exemple implémenter les booléens en réutilisant les naturels avec 0 et *succ* où l'unique opérande de *succ* serait la constante 0, ou la spécification des n-uplet par l'implémentation des (m)-uplet (où $m > n$. Par exemple, il peut y avoir avantage à ce que $n + 1$ soit pair).

Exemple 5.3: Soient les spécifications classiques des booléens et des naturels :

```
Type Bool                          Type Nat
sorts                              sorts
    bool                               nat
constructors                       constructors
    true : -> bool                     zero : -> nat
    false : -> bool                    succ : nat -> nat
operators                          operators
⋮                                  ⋮
end                                end
```

Soit une implémentation Ada de la sorte *nat* par le type `integer` :

$\Delta_{Nat}(nat) = integer$

$\mathcal{E}(zero) = [\,]\{0\}$ $\qquad$ $\mathcal{E}(succ) = [x]\{x+1\}$

$In(zero) = <>$ $\qquad$ $In(succ) = < \pi_1 >$

$Out(zero) = <>$ $\qquad$ $Out(succ) = < \phi >$

La réutilisation d'une implémentation des constructeurs des naturels pour ceux des booléens se fait en fournissant le lien d'implémentation abstraite :

$\Delta^{I^+}_{Bool}(bool) = nat$

$\mathcal{E}^{I^+}_{Bool}(true) = [\,]\{zero\}$ $\qquad$ $\mathcal{E}^{I^+}_{Bool}(false) = [x]\{succ(x)\}$

$In^{I^+}_{Bool}(true) = <>$ $\qquad$ $In^{I^+}_{Bool}(false) = < 0 >$

$Out^{I^+}_{Bool}(true) = <>$ $\qquad$ $Out^{I^+}_{Bool}(false) = < \phi >$

Les constructeurs du type algébrique *Bool* peuvent alors réutiliser l'implémentation des constructeurs du type algébrique *Nat* par le type Ada `integer`, via la composition $\Delta \circ \Delta^{I^+}$. Nous avons choisi la constante 0 comme constante Ada de type $(\Delta_{Nat} \circ \Delta^{I^+}_{Bool})(bool) = integer$, et nous obtenons :

$\Delta_{Bool}(bool) = integer$

$$\begin{aligned}\mathcal{E}(true) &= \mathcal{E}_{Nat}(\mathcal{E}^{I^+}_{Bool}(true)) \\ &= \mathcal{E}_{Nat}(zero) = [\,]\{0\}\end{aligned}$$

$$\begin{aligned}In(true) &\\ &= In_{Nat}(zero) \circ In^{I^+}_{Bool}(true) \\ &= <> \circ <> = <>\end{aligned}$$

$$\begin{aligned}Out(true) &\\ &= Out_{Nat}(zero) \circ Out^{I^+}_{Bool}(true) \\ &= <> \circ <> = <>\end{aligned}$$

$$\begin{aligned}\mathcal{E}(false) &= \mathcal{E}_{Nat}(\mathcal{E}^{I^+}_{Bool}(false)) \\ &= \mathcal{E}_{Nat}(succ) = [x]\{x+1\}\end{aligned}$$

$$\begin{aligned}In(false) &\\ &= In_{Nat}(succ) \circ In^{I^+}_{Bool}(false) \\ &= < \pi_1 > \circ < 0 > \\ &= < 0 >\end{aligned}$$

$$\begin{aligned}Out(false) &\\ &= Out_{Nat}(succ) \circ Out^{I^+}_{Bool}(false) \\ &= < \phi > \circ < \phi > = < \phi >\end{aligned}$$

Le terme $false$ sera donc dérivé en $\langle\!|0+1|\!\rangle$

■

Chapitre 6

Flots et mutations

Un flot de données est l'association d'une variable x et d'une suite de calculs appliqués successivement à cette variable, dans laquelle est rangé le résultat de chaque calcul.

Définition 6.1: (Flot)
Un flot est constitué d'une variable x et d'une séquence de p nœuds $\mathcal{N}_{a_1}, \ldots, \mathcal{N}_{a_p}$ tels que :

- $\forall i \in [1 \ldots p-1] \cdot (a_i \in \mathcal{N}_{a_{i+1}}.\, operandes \ \wedge \ x \in \mathcal{N}_{a_i}.\, destination)$
- l'*origine* $\mathcal{N}_{a_1}$ du flot est un nœud qui code la variable x (une feuille), ou dans lequel x reçoit une valeur qui n'est pas une modification de sa propre valeur :

$$\begin{array}{l} \mathcal{N}_{a_1}.variable = x \\ \vee \ \exists j \cdot \left(\begin{array}{l} x = \mathcal{N}_{a_1}.destination_j \\ \wedge \ x \neq \mathcal{N}_{a_1}.Out_j \end{array} \right) \\ \vee \ \exists j \cdot \left(\begin{array}{l} x = \mathcal{N}_{a_p}.Out_j \\ \wedge \ Out\text{-}to\text{-}In(\mathcal{N}_{a_1}.operateur)_j = \phi \end{array} \right) \end{array}$$

- l'*extrémité* $\mathcal{N}_{a_p}$ du flot est soit le nœud $\mathcal{N}_1$ (il n'est pas un opérande), soit tel que $x \notin \mathcal{N}_{a_p}.\, destination$.

■

Nous nous intéressons aux flots dont l'extrémité finale est le nœud $\mathcal{N}_1$ qui code le terme T. Le terme T est le terme à dériver pour obtenir le corps de programme de l'implémentation de l'opérateur d'intérêt.

6.1 Dérivation avec les destinations des sous-termes fixées

L'exemple 4.11 page 105 montre un cas où les affectations finales aux paramètres formels de retour de l'opérateur d'intérêt peuvent être (dans certains cas) évitées : l'un

des deux remplacements est trivial :

$$\boxed{\begin{array}{l} duplication(z, z_2); \\ renverse(z_2); \\ v_1 := concat_3(x, y, z_2); \\ concat(v_1, t); \\ concat(t, x); \\ concat(x, z); \\ v_2 := tri(t); \\ duplication(v_1, z); \\ duplication(v_2, t); \end{array}} \quad \text{devient} \quad \boxed{\begin{array}{l} duplication(z, z_2); \\ renverse(z_2); \\ v_1 := concat_3(x, y, z_2); \\ concat(v_1, t); \\ concat(t, x); \\ concat(x, z); \\ t := tri(t); \\ duplication(v_1, z); \end{array}}$$

Le remplacement de

$$v_1 := concat_3(x, y, z_2);\ concat(v_1, t);$$

par

$$z := concat_3(x, y, z_2);\ concat(z, t);$$

serait incorrect. En effet, **z** serait modifié alors que sa valeur est nécessaire deux instructions plus loin.

Il n'est pas seulement avantageux d'éviter des affectations finales aux paramètres de retour : la dérivation par anticipation d'un nœud hors du code du nœud dont il est l'opérande, nécessite la connaissance de la destination de son résultat. Cela permet l'affectation des résultats de fonctions à des variables autres que des variables intermédiaires et l'utilisation des variables correctes comme paramètres effectifs résultats. Cela évite des affectations supplémentaires lorsque les destinations sont connues ou calculables.

Nous allons donc modifier notre algorithme de calcul des destinations des résultats défini en section 4.2 page 85, ainsi que la manière dont sont calculées les utilisations et modifications de variables. En effet, la localisation des conflits ne peut plus se faire comme il est décrit en section 4.1.2 page 73 :

Les modifications précoces de paramètres retour doivent être détectées au niveau concret

À partir de l'équation définissant l'opérateur d'intérêt, et pour une dérivation vers une procédure, nous obtenons l'interface Ada (c'est-à-dire le profil de la procédure ou fonction), ainsi que la liste de ses paramètres formels qui doivent contenir à la fin de son exécution, une représentation de la valeur du terme en partie droite de l'équation à dériver (dans le cas d'une procédure).

Si l'opérateur *op* de T est dérivé en un appel de procédure modifiant ses paramètres, en connaissant les variables qui doivent contenir le résultat, nous pouvons (règle 3.5 page 69) calculer les variables qui doivent être en paramètre effectif à l'appel de cette procédure. Nous connaissons ainsi pour chaque sous-terme opérande de *op*, quelle doit être la destination de son résultat. Pour des sous-termes dont l'opérateur est implémenté par une procédure, et dont la destination du résultat a été ainsi déterminée, il nous

est possible d'utiliser cette destination comme paramètre effectif en mode `out` (ou `in out`, et dans ce cas nous lui affectons au préalable la valeur de l'opérande si celui-ci est différent).

En conséquence, nous nous retrouvons avec des paramètres formels de l'opérateur d'intérêt, qui sont modifiés au niveau concret par le code dérivé d'un opérateur de T dont ils ne sont pas des opérandes au niveau abstrait.

Exemple 6.1: Soit l'équation définissant l'opérateur f :

$$f(x) ==> append(cdr(x), sort(x))$$

Le lien de dérivation de f et les liens d'implémentation des opérateurs en partie droite de l'équation sont les suivants :

- f devient une procédure `F` qui modifie son unique paramètre,
- *append* devient une procédure qui modifie son premier paramètre :

  $\mathcal{E}(append) = [x, y]\{$`APPEND(x,y)`$\}$,
  $In(append) = < \pi_1, \pi_2 >$,
  $Out(append) = < \pi_1, \phi >$,

- *cdr* devient une simple fonction unaire : `CDR`,
- *sort* devient une procédure à 2 paramètres dont le premier est `in` et le second `out` :

  $\mathcal{E}(sort) = [x, y]\{$`SORT_TO(x,y)`$\}$,
  $In(sort) = < \pi_1, \phi >$,
  $Out(sort) = < \phi, \pi_1 >$.

Les sous-termes $sort(x)$ et $cdr(x)$ ne modifient pas leur opérande, nous pouvons exécuter leurs textes dérivés dans l'ordre que nous voulons. Si nous prenons une évaluation de gauche à droite et en profondeur d'abord, nous obtenons une séquence :

$$[x1 = sort(x); x2 = cdr(x); x3 = append(x1, x2)]$$

que nous pouvons traduire à la syntaxe Ada :

```
procedure F(X: in out t) is
begin
  X1 := CDR(X);
  SORT_TO(X,X2);
  APPEND(X1,X2);
  X := X1 ;
end F;
```

Faire hériter chaque calcul de la destination des résultats permet d'éviter des variables intermédiaires. La variable `X`, paramètre formel, doit contenir le résultat en fin d'exécution du corps de la procédure implémentant f. L'opérateur en racine du sous-terme dont `X` est la destination est la procédure `APPEND` qui a son premier paramètre en mode

`in out`, nous en déduisons que `X` peut être le premier paramètre effectif de `APPEND` et donc être la variable affectée du résultat du `CDR` :

```
procedure F(X: in out t) is
begin
  X := CDR(X);
  SORT_TO(X,X2);
  APPEND(X,X2);
end F;
```

Ce qui donne un résultat erroné car `X` est modifié avant son utilisation par `SORT_TO`. Nous voyons qu'un calcul qui n'est dérivé qu'en une application d'une fonction, comme $cdr(x)$, modifie une variable si nous faisons hériter les nœuds de la destination des résultats, jusqu'à ceux qui sont les plus profonds.

Nous devons donc, soit déterminer différemment les variables modifiées avant de faire l'optimisation "calcul utilisateur avant calcul modificateur" et réordonner l'évaluation des sous-termes, soit ne pas faire "descendre" complètement dans le terme dérivé la variable modifiée qui doit contenir le résultat d'un sous-terme :

```
procedure F(X: in out t) is
begin
  SORT_TO(X,X2);
  X := CDR(X);
  APPEND(X,X2);
end F;
```

```
procedure F(X: in out t) is
begin
  X1 := CDR(X);
  SORT_TO(X,X2);
  X := X1;
  APPEND(X,X2);
end F;
```

■

Nous allons donc modifier les différentes étapes de la résolution des conflits décrite au chapitre 4.

6.1.1 Calcul de la destination du résultat de chaque sous-terme

Nous décorons chaque nœud $\mathcal{N}$ avec $\mathcal{N}.destination$. La destination du résultat est un produit cartésien formé de variables ou de la constante ϕ. Cette décoration se fait par un parcours de l'arbre représentant T (récursivement tous les nœuds $\mathcal{N}_i.operandes$ de chaque nœud $\mathcal{N}_i$, en commençant par $\mathcal{N}_1$) où chaque attribut *destination* est calculé en fonction de l'attribut *destination* du nœud dont il est l'opérande de la manière suivante :

Calcul de l'attribut *destination* de la racine $\mathcal{N}_1$ de l'arbre codant T.
Ce sont les paramètres formels où seront stockés les résultats en fin d'exécution du code

dérivé pour T.

Pour

- $op(\alpha_1, \ldots, \alpha_n)$ ==> $\mathrm{T}_{[\alpha_1, \ldots, \alpha_n]}$ l'équation définissant l'opérateur $op : (s_1, \ldots, s_n) \text{ -> } (s_{n+1}, \ldots, s_{n+m})$,
- $(x_1, \ldots, x_n)$ et $(x_{n+1}, \ldots, x_{n+m})$ des variables de types respectifs $(\Delta_S(s_1), \ldots, \Delta_S(s_n))$ et $(\Delta_S(s_{n+1}), \ldots, \Delta_S(s_{n+m}))$,

d'après la figure 3.3 page 61, les paramètres formels de la fonction ou procédure dérivée sont :

$$In(op)(x_1, \ldots, x_n) \oplus_L Out(op)(x_{n+1}, \ldots, x_{n+m})$$

et les résultats de la dérivation de $\mathrm{T}_{[x_1, \ldots, x_n]}$ ont pour destination, dans le cas d'une procédure :

$$Out(op)^{-1}(In(op)(x_1, \ldots, x_n) \oplus_L Out(op)(x_{n+1}, \ldots, x_{n+m}))$$

et dans le cas d'une fonction, un produit cartésien de m éléments égaux à ϕ.

Héritage de l'attribut *destination* pour les nœuds autres que $\mathcal{N}_1$.
Soit un nœud $\mathcal{N}$ tel que $\mathcal{N}.operateur = c$ avec $c : (s_1, \ldots, s_n) \text{ -> } (s_{n+1}, \ldots, s_{n+m})$ et $\mathcal{N}.destination = (dest_1, \ldots, dest_m)$. Le calcul des destinations des nœuds (qui ne codent pas des variables) opérandes du nœud $\mathcal{N}$, sont calculées de la manière suivante :

Pour un nœud $\mathcal{N}$ ayant autant d'opérandes que de sortes du domaine de son opérateur :
$\forall j \in \mathcal{N}.operandes \cdot \mathcal{N}_j.destination = (Out\text{-}to\text{-}In(c)(dest_1, \ldots, dest_m))_j$

Pour un nœud $\mathcal{N}$ ayant un unique opérande représentant un terme dont la sorte est un produit cartésien de sortes :
$\mathcal{N}.operateur : (s_1 \ldots s_n) \text{ -> } (s_{n+1} \ldots s_{n+m})$ et $\mathcal{N}.operandes = (id)$,
alors $\mathcal{N}_{id}.destination = \mathcal{N}.destination$

Pour un nœud $\mathcal{N}$ qui code un if-then-else :
Pour $\mathcal{N}.operandes = (id_1, id_2, id_3)$ nous avons :

- $\mathcal{N}_{id_1}.destination = \phi$, pour le test booléen,
- $\mathcal{N}_{id_2}.destination = \mathcal{N}.destination$, pour l'alternative positive,
- $\mathcal{N}_{id_3}.destination = \mathcal{N}.destination$, pour l'alternative négative.

Une fois les destinations héritées, nous calculons les destinations, ou éléments de destination, qui n'ont pas été précisées, par le calcul à partir des opérandes défini dans la section 4.2 page 85, avec la modification suivante :

- pour tout nœud $\mathcal{N}$ différent d'un "*if-then-else*", la destination calculée (non héritée) est rangée dans $\mathcal{N}.Out$. Dans $\mathcal{N}.\ destination$ nous rangeons :

$$(\mathcal{N}.\ destination \boxplus_L \mathcal{N}.Out) \oplus_L (v_1, \ldots, v_m)$$

avec $(v_1, \ldots, v_m)$ des variables fraîches.
Nous ajoutons les post-affectations $duplication(x, y)$ telles que :

$$\exists i \in [1 \ldots m] \cdot \left(\begin{array}{l} x = \mathcal{N}.Out_i \\ \wedge\ y = \mathcal{N}.\,destination_i \\ \wedge\ x \neq y \end{array} \right)$$

- pour un nœud $\mathcal{N}$ qui code un "*if-then-else*", la destination commune aux sous-termes *then* et *else* est calculée par (avec les notations utilisées dans la section 4.2) :

$$((((\mathcal{N}.\,destination \boxplus_L \mathtt{D}^{t,pf}) \boxplus_L \mathtt{D}^{e,pf}) \boxplus_L \mathtt{D}^{t}) \boxplus_L \mathtt{D}^{e}) \oplus_L (v_1, \ldots, v_m)$$
ou
$$((((\mathcal{N}.\,destination \boxplus_L \mathtt{D}^{e,pf}) \boxplus_L \mathtt{D}^{t,pf}) \boxplus_L \mathtt{D}^{e}) \boxplus_L \mathtt{D}^{t}) \oplus_L (v_1, \ldots, v_m)$$

avec $(v_1, \ldots, v_m)$ des variables fraîches.
La destination peut également être calculée par une recherche exhaustive de la destination :

$$(\mathcal{N}.\,destination \boxplus_L \mathtt{D}^{complet}) \oplus_L (v_1, \ldots, v_m)$$

dont le coût en post-affectations ajoutées aux nœuds *then* et *else* est minimum, avec $\mathtt{D}^{complet}$ une combinaison de m variables (pour compléter la destination commune) de $\mathtt{D}^t \cup \mathtt{D}^e$. Les post-affectations à ajouter aux nœuds *then* et *else* sont calculées comme dans la section 4.2.

6.1.2 Calcul des conflits d'accès aux variables

La résolution des conflits s'effectue avec le même algorithme que dans le chapitre 4, mais les modifications sont calculées différemment. En effet, l'héritage des destinations fait qu'un paramètre formel de l'opérateur d'intérêt peut être en paramètre effectif de sortie uniquement dans le code généré. Nous devons donc compter comme nœuds modificateurs d'une variable x certains nœuds dont x ne fait pas partie des opérandes dans le terme algébrique représenté. Nous comptons comme modificateur d'une variable x un nœud qui est l'origine d'un flot associé à x. La résolution des conflits d'accès à x existants avec les nœuds d'un flot associé à x sont résolus par la résolution des conflits d'accès à x existants avec le nœud origine du flot.

La variable x est dite modifiée par le texte dérivé du nœud $\mathcal{N}$ si :

- x est opérande de $\mathcal{N}$ et modifiée par le texte dérivé de $\mathcal{N}$,
- x est un paramètre en sortie uniquement du texte dérivé pour $\mathcal{N}$.

Le calcul défini page 92 de l'attribut `M` d'un nœud $\mathcal{N}$, qui code les modifications de tous les nœuds du sous-arbre dont $\mathcal{N}$ est racine, devient :

$$\mathtt{M} = \bigcup_{i\in 1..n} \mathtt{M}_i \cup \left\{ (\vec{v},\mathcal{N}_o) \;\middle|\; \forall v \in \vec{v} \cdot \begin{array}{ll} & \begin{pmatrix} v \in \mathcal{N}_o.operandes \\ \wedge\ Var(v) \\ \wedge\ v \in P^{op}_{In\text{-}Out}(\mathcal{N}_o.operandes) \\ \wedge\ v \in Param_{Out} \cup Param_{In\text{-}Out} \end{pmatrix} \\ \vee\ \exists j \cdot & \begin{pmatrix} Var(v) \\ \wedge\ v \in Param_{Out} \cup Param_{In\text{-}Out} \\ \wedge\ v = \mathcal{N}_o.destination_j \\ \wedge\ v \neq \mathcal{N}_o.Out_j \end{pmatrix} \\ \vee\ \exists j \cdot & \begin{pmatrix} Var(v) \\ \wedge\ v \in Param_{Out} \cup Param_{In\text{-}Out} \\ \wedge\ v = \mathcal{N}_o.Out_j \\ \wedge\ Out\text{-}to\text{-}In(op)_j = \phi \end{pmatrix} \end{array} \right\}$$

Les calculs des attributs d'utilisation et des arcs $\mathcal{R}_<$ sont identiques à ceux définis page 92. La suite de la dérivation est identique.

6.1.3 Minimisation optimale des duplications

Nous avons au chapitre 4 montré une méthode qui résoud les conflits d'accès aux variables, en effectuant un minimum de duplication de variables. Cette méthode est optimale si l'on ne considère pas les post-affectations (affectations finales aux paramètre formels de retour d'une procédure, sous-termes *then* et *else* de manière à avoir la même destination, ou variable locale définie par un *let*).

La variante que nous avons décrite dans ce chapitre montre comment faire hériter les calculs des destinations des résultats. Mais cette méthode n'est pas optimale si l'on effectue cet héritage de manière systématique pour tous les calculs :

Exemple 6.2: Soit l'opérateur *op* : $(seqnat, seqnat)$ -> $(seqnat, seqnat)$ dont l'équation est la suivante :

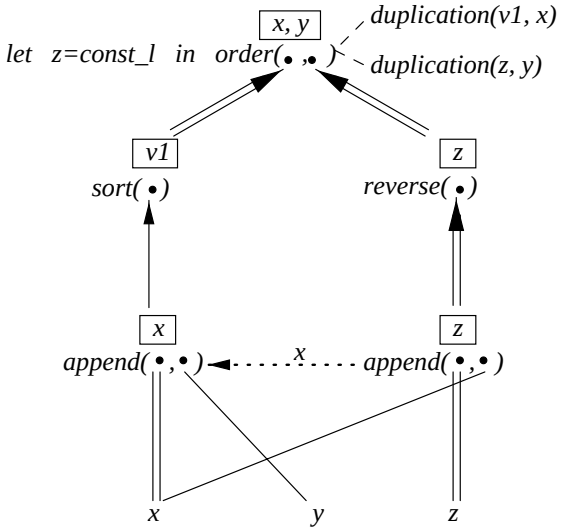
$$op(x,y) ==> let\ z = const\_l\ in\ order(sort(append(x,y)), reverse(append(z,x)))$$

Soit le lien de dérivation de *op* : $\mathcal{E}(op) = [x,y]\{\mathtt{OP(x,y)}\}$
$In(op) = < \pi_1, \pi_2 >$
$Out(op) = < \pi_1, \pi_2 >$
Les résultats de la procédure `OP(x, y: in out listint);` doivent être contenus dans `x` et `y`.

L'opérateur *sort* est implémenté par une fonction, *reverse* par une procédure ayant un unique paramètre en entrée-sortie, *order* par une procédure dont les deux paramètres sont en entrée-sortie, et *append* comme nous le connaissons déjà. Nous obtenons avec la méthode décrite au chapitre 4 et avec celle que nous présentons maintenant, les graphes

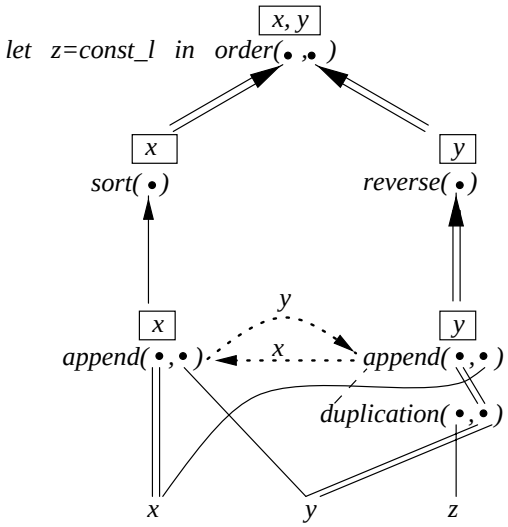
suivants d'ordonnancement des calculs :

Synthèse des destinations



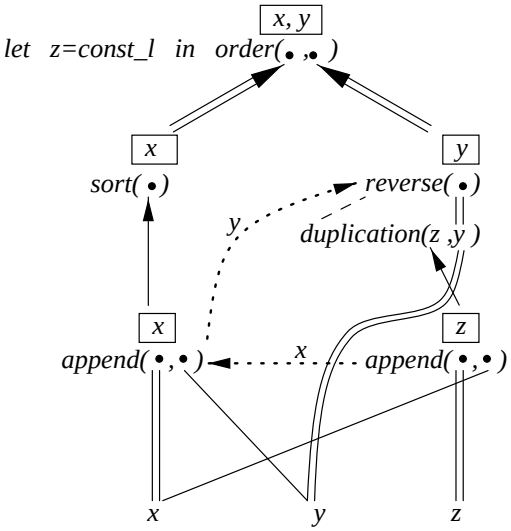
Dans le cas de la synthèse des destinations (chapitre 4), le résultat de l'application d'une fonction est stocké dans une variable libre auxiliaire, et ce n'est qu'en fin de calcul que nous affectons les résultats aux paramètres formels de retour.

Héritage des destinations



Avec l'héritage des destinations des résultats des calculs intermédiaires, ces affectations finales peuvent être évitées, comme celles parfois nécessaires à l'obtention d'une destination commune à des sous-termes *then* et *else*. Mais ces affectations précoces peuvent engendrer des conflits (ou également en éviter, ce qui aurait été le cas de notre exemple si le terme $append(z, x)$ avait été en conflit d'accès à z).

Solution optimale



La solution optimale peut se trouver dans un compromis entre les deux méthodes. Nous devons alors évaluer le coût total pour toutes les positions où le choix entre la synthèse et l'héritage de la destination fait une différence en coût de duplication (appel de fonction ou de procédure avec un paramètre en sortie uniquement, création ou suppression d'un conflit d'accès).

■

6.2 Mutations

Les mutations de données sont des modifications partielles d'un objet structuré. Nous nous plaçons au niveau du domaine abstrait, où le calcul d'un terme t_2 qui ne diffère que partiellement d'un terme t_1, est défini par une construction de t_2 à partir de composants de t_1. Ces composants sont obtenus, soit par filtrage ("pattern-matching") avec les paramètres formels en partie gauche de l'équation à dériver, soit par utilisation explicite de sélecteurs algébriques.

Dans le cas du pattern-matching :

$$op(x_1, \ldots, x_{i-1}, c(x'_1, \ldots, x'_m), x_{i+1}, \ldots, x_n) ==> \\ \mathrm{T}[x_1, \ldots, x_{i-1}, x'_1, \ldots, x'_m, x_{i+1}, \ldots, x_n]$$

doit être transformé, pour être dérivé, en :

$$op(x_1, \ldots, x_n) ==> \mathrm{T}_{(x_1, \ldots, x_{i-1}, sel^c_1(x_i), \ldots, sel^c_m(x_i), x_{i+1}, \ldots, x_n)}$$

et les sélecteurs $sel^c_1, \ldots, sel^c_m$ qui peuvent être utilisés explicitement dans T sont définis par :

$$\forall i \in [1 \ldots n] \cdot (sel^c_i(c(x_1, \ldots, x_n) = x_i)$$

Soit un nœud $\mathcal{N}$ qui code un terme t de la forme :

$$c(sel^c_1(x), \ldots, sel^c_{i-1}(x), t', sel^c_{i+1}(x), \ldots, sel^c_m(x))$$

avec c un constructeur, et dont la destination du nœud est la variable x. Le terme t est avantageusement remplacé par le terme $muter^c_i(x, t')$, qui correspond au remplacement de l'opérande de x en position i par t', sachant que le constructeur de x est c.
Soit $\Delta^{\#}(t') = ([\![p^{t'}]\!], \langle\!|f^{t'}|\!\rangle)$, et $\Delta_D(muter^c_i) = (\mathcal{E}(muter^c_i), In(muter^c_i), Out(muter^c_i))$ avec :

$$\mathcal{E}(muter^c_i) = [x, v]\{\texttt{MUTER\_}c\texttt{\_}i(x, v)\} \\ In(muter^c_i) = < \pi_1, \pi_2 > \\ Out(muter^c_i) = < \pi_1 >$$

nous obtenons alors :

$$\Delta^{\#}(t) = ([\![p^{t'}; \texttt{MUTER\_}c\texttt{\_}i(x, f^{t'});]\!], \langle\!|x|\!\rangle)$$

ou, si le sélecteur sel^c_i est une expression-gauche, avec $\Delta^{\#}(sel^c_i(x)) = ([\![\,]\!], \langle\!|f^{sel^c_i}(x)|\!\rangle)$:

$$([\![p^{t'}; f^{sel^c_i}(x) := f^{t'};]\!], \langle\!|x|\!\rangle)$$

Dans le cas d'un terme t similaire, mais où plusieurs opérandes de c, chacun d'indice i ne sont pas égaux à $sel^c_i(x)$, il est aisé de définir comment t est dérivé en une séquence d'applications de mutateurs.

Conclusion

Nous avons dans ce chapitre proposé une introduction à une approche "flot de données" des calculs : après une définition formelle d'un flot, nous avons montré une méthode de dérivation qui est une variante de la méthode proposée au chapitre 4 : les destinations des calculs intermédiaires sont héritées, lorsque la destination du résultat final, quand s'agit d'une procédure, est connue. Nous avons enfin proposé une modification préalable du terme algébrique T à dériver, lorsqu'il contient une construction d'un terme t' à l'identique de t, sous-terme d'un opérande de l'opérateur d'intérêt, à l'exception de certaines positions. Cette construction est alors transformée en une modification en-place du sous-terme d'origine, aux positions où t et t' diffèrent.

Ces deux variantes apportent :

- pour l'héritage des destinations :
  - dans certains cas (procédures et sous-termes *if-then-else* avec des calculs intermédiaires fonctionnels) un coût optimal (minimum) en duplications, affectations finales et post-affectations comprises,
  - la possibilité d'avoir des flots de données supplémentaires dont les variables qui les constituent sont celles qui doivent contenir (ou composent) le résultat final.
- pour l'introduction des mutations :
  - une optimisation des calculs, en remplaçant la construction d'un résultat à partir de sélections de composants, par une modification partielle de la variable dans laquelle ces composants étaient sélectionnés,
  - la possibilité d'obtenir des flots de données supplémentaires qui aboutissent au résultat final, en remplaçant des sous-termes de T par des modifications partielles d'une variable d'un flot.

L'objectif suivi par cette approche, est d'obtenir un flot de données pour tout paramètre formel en entrée-sortie : nous pouvons supposer que l'idée sous-jacente à l'introduction d'un paramètre de ce mode est une modification partielle d'une donnée. Une orientation dans la suite des travaux sur ce sujet pourrait être le calcul de la différence entre un opérande et le sous-terme résultat qui lui correspond par le lien d'implémentation. Nous pourrions alors générer les instructions qui effectuent les modifications liées à cette différence.

Chapitre 7

Outil

7.1 Description de l'outil

Un prototype est réalisé en CAML. Il implémente une version simplifiée de l'algorithme des chapitres 3 et 4 pour un sous-langage très restreint du langage LPG : l'algorithme ne traite pas différemment les paramètres formels en entrée ou en entrée-sortie, il n'y a pas de calcul de destination unique d'un "*if-then-else*". Ces deux limitations sont dues au fait que les pré- et post-affectations ne sont pas implémentées. De plus, les opérateurs définis ne peuvent avoir un produit cartésien de sortes en co-domaine.

En entrée, le lien d'implémentation et l'arbre abstrait doivent être fournis sous la forme d'une structure en CAML. Les arcs supplémentaires sont alors calculés, les cycles éventuels déterminés ainsi que les coupures de coût minimum. En sortie, la liste des noeuds du graphe est fournie dans l'ordre de l'exécution du code dérivé de chaque noeud dans le domaine concret.

7.2 Syntaxe du lien d'implémentation

Exemple de lien en Bibliothèque entre une Spécification Algébrique et une de ses implémentations en Ada :

```
Files
    Seq.lpg     >-> listes.ads, listes.adb
Types
    Seq         >-> LISTES
Sorts
    seq         >-> liste
Constructors
    nil         >-> creation_vide(): out($1)
    <+          >-> creation($1:in, $2:in): out($3)
Testors
    nil         >-> est_vide(_:in)
    <+          >-> not(est_vide(_:in))
Selectors
    <+          >-> acces_element(_:in), acces_suivant(_:in)

Operators
    head        >-> acces_element($1:in): out($2)
    tail        >-> acces_suivant($1:in): out($2)
```

```
    +           >-> concatener($1:in,$2:in): out($3)
    +>          >-> ajout_en_fin($1:in out($3),$2:in)
Mutators
    <+          >-> changer_element(_:in out(_),$1:in) ,
                    changer_suivant(_:in out(_),$2:in)
Specials
    =           >-> equals($1:in,$2:in):out(Bool)
    duplicate   >-> copie(_:in):out(_)
```

Un autre exemple d'implémentation du constructeur de listes `<+`, cette fois par une procédure dont la liste en deuxième paramètre est modifiée pour retourner la liste résultat :

```
Constructors
    <+          >-> ajout_en_tete($1:in, $2:in out($3))
```

Pour lever l'ambiguïté d'une surcharge dans la même unité, nous pouvons préciser la sorte exacte de l'opérateur `<+` :

```
( <+ : elem,seq[elem] -> seq[elem] )  >-> creation($1:in, $2:in): out($3)
```

Syntaxe :

Soit $\mathcal{R}$ la relation de représentation entre une algèbre A et une unité U Ada qui l'implémente : $\mathcal{R}(A, U)$ signifie que l'unité U *implémente* l'algèbre A.

R est la relation de réalisation. Elle est satisfaite par le terme fermé t de A et le terme t' instancié P lorsque t' est l'implémentation de t.

Files — est l'association du fichier contenant la spécification algébrique au doublet spécification-corps de son implémentation en Ada.

Types — est l'association de l'unité LPG de la bibliothèque au package Ada qui l'implémente.

Sorts — est l'association à chaque sorte LPG d'un type qui l'implémente.

Constructors — est l'association d'un constructeur c de LPG à une fonction ou procédure qui l'implémente en Ada.
Les sortes du domaine suivi du co-domaine de c sont indexées, ainsi la correspondance entre les paramètres de c' et les paramètres de c est indiquée, avec le mode de chaque paramètre de c'. Exemples :

```
cons  >->  creation1($1:In, $2:In):Out($3)
cons  >->  creation2($2:In-Out($3), $1:In)
```

signifie que si R est la relation de représentation entre tout terme LPG et tout terme Ada, nous pouvons définir en logique de Hoare : `(pfpré) {expr} (pFpost)`

$$(R(x,x') \wedge R(y,y'))\ \{z' := creation1(x',y')\}\ (R(cons(x,y),z'))$$

$$(R(x,x') \wedge R(y,y'))\ \{creation2(y',x')\}\ (R(cons(x,y),y')$$

Soient n, m et n', m' les cardinalités des domaines et co-domaines respectifs de l'opérateur c et de la procédure ou fonction c'. ($m > 1$ pour un co-domaine de c produit cartésien de sortes, $m' = 1$ si c' est une fonction et $m' = 0$ pour une procédure).
La forme générale du lien d'implémentation est :

$$\mathcal{E}(c) = [v_1, \ldots, v_{n'}]\{c(v_1, \ldots, v_{n'})\}$$
$$In(c) = < P_1^{In(c)}, \ldots, P_{n'}^{In(c)} >$$
$$Out(c) = < P_1^{Out(c)}, \ldots, P_{n'}^{Out(c)} >$$

Il est définit de manière syntaxique par :
$\forall k \in [1 \ldots n'] \cdot$

$$\begin{pmatrix} \texttt{c >-> c'}(\ldots, par_k \equiv \texttt{\$}i\texttt{:in}, \ldots)\ldots & \Rightarrow & P_k^{In(c)} = \pi_i \\ \texttt{c >-> c'}(\ldots, par_k \equiv \texttt{\$}i\texttt{:in out(\$}j\texttt{)}, \ldots) & \Rightarrow & P_k^{In(c)} = \pi_i \\ \texttt{c >-> c'}(\ldots, par_k \equiv \texttt{out(\$}j\texttt{)}, \ldots) & \Rightarrow & P_k^{In(c)} = \phi \end{pmatrix}$$

$\forall k \in [1 \ldots n'] \cdot$

$$\begin{pmatrix} \texttt{c >-> c'}(\ldots)\texttt{:out(\$}j\texttt{)} & \Rightarrow & P_k^{Out(c)} = \phi \\ \texttt{c >-> c'}(\ldots, par_k \equiv \texttt{\$}i\texttt{:in out(\$}j\texttt{)}, \ldots) & \Rightarrow & P_k^{Out(c)} = \pi_{j-n} \\ \texttt{c >-> c'}(\ldots, par_k \equiv \texttt{out(\$}j\texttt{)}, \ldots) & \Rightarrow & P_k^{Out(c)} = \pi_{j-n} \end{pmatrix}$$

Testors — est l'association à chaque constructeur c de LPG d'une fonction booléenne dont l'unique paramètre en entrée est noté par le caractère "_". L'évaluation de la fonction donne un résultat booléen vrai lorsque le paramètre est la représentation d'un terme construit par le constructeur c.

Selectors — est l'association à tout constructeur c de domaine non vide de cardinal n, d'une séquence de n fonctions dont l'unique paramètre est noté par le caractère "_". La $i^{ème}$ fonction, appliquée à un terme du type représentant la sorte du constructeur c, renvoie la valeur qui représente le $i^{ème}$ terme LPG nécessaire pour la construction par c du terme représenté par le terme testé.

$$c \quad \texttt{>->} \quad s'_1(\_ : In)\texttt{,} \ \ldots\texttt{,} \ s'_n(\_ : In)$$
$$(R(c(x_1..x_n), x'))\ \{y' := s'_i(x')\}\ (R(x_i, y')),\ \forall i \in \{1..n\}$$

Operators — est l'association d'un opérateur op de LPG à une fonction ou procédure fp' l'implémentant en Ada.
La correspondance entre les paramètres de fp' et les paramètres de op, le mode de chaque paramètre de fp', et quel paramètre en entrée est modifié par le résultat si tel est le cas, sont indiqués de la même manière que pour les constructeurs.

Mutators est l'association à tout constructeur c de domaine non vide de cardinal n, d'une séquence de n fonctions ou procédures m'_i appelées mutateurs (les procédures doivent n'avoir qu'un seul paramètre en mode Out ou In–Out). Chaque mutateur m'_i a 2 paramètres en entrée, l'un noté "_" du type représentant la sorte du constructeur c, et l'autre noté "`$i`" du type représentant la sorte du $i^{ème}$ paramètre du constructeur c. L'unique paramètre en sortie du mutateur m'_i noté également "_" est du type représentant la sorte du constructeur c.
m'_i satisfait la définition suivante :

c `>->` $m'_1(\_ : In, \$1 : In) : Out(\_)$`,...,` $m'_n(\_ : In, \$n : In) : Out(\_)$

ou

c `>->` $m'_1(\_ : In\text{-}Out(\_), \$1 : In)$`,...,` $m'_n(\_ : In\text{-}Out(\_), \$n : In)$

avec

$(R(c(x_1..x_n), x'))\{y' := m'_i(x', a)\}(R(c(x_1, .., x_{i-1}, a, x_{i+1}, .., x_n), y') \wedge x' = y')$`,` $\forall i \in \{1..n\}$

ou

$\forall i \in \{1..n\} \cdot$
$(R(c(x_1..x_n), x'))\{m'_i(x', a)\}(R(c(x_1, .., x_{i-1}, a, x_{i+1}, .., x_n), x'))$

Specials est l'association de plusieurs opérateurs spéciaux à leur équivalent en Ada. Ces opérateurs sont :

l'opérateur booléen d'égalité entre 2 termes de même sorte

l'opérateur de recopie d'un terme qui, en Ada, reconstruit un terme ayant même valeur que le paramètre passé en entrée (représentant le même terme LPG) mais sans partage de donnée.

Exemple 7.1: Le fichier qui décrit le lien d'implémentation de l'opérateur *op* de l'exemple 3.8 page 62 contient :

```
op      >-> OP($1:in out($2), $2:in, $3:in out($1), $4:in out($3))
```

Le lien d'implémentation (et de dérivation de *op*) est le suivant:

$\mathcal{E}(op) = [x, y, z, t]\{$`OP(x,y,z,t)`$\}$,
$In(op) = < \pi_1, \pi_2, \pi_3, \pi_4 >, Out(op) = < \pi_2, \phi, \pi_1, \pi_3 >$

■

7.3 Exemples – résultats

Cette section montre le résultat d'une session de dérivation d'un terme algébrique codé en un arbre abstrait vers une séquence d'instructions Ada codées en une liste de nœuds.

Dans la syntaxe que nous avons proposée, le lien d'implémentation des différents opérateurs qui apparaissent dans le terme à dériver, est décrit par :

```
Operators
    f         >->  F($3:in, $2:in, $1:in): out($4)
    g         >->  G($1:in out($3), $2:in)
    +         >->  H($1:in): out($2)
    +>        >->  P($1:in out($2))
```

Nous introduisons le lien d'implémentation dans sa forme interne CAML :

```
#modif_g_param_mapping env_global
    (map__add (Oper("F",Profil([],[])))
              ((map__add 1 3
                (map__add 2 2
                 (map__add 3 1 empty_int_mapping))),
               (map__add 4 4 empty_int_mapping))

    ( map__add (Oper("G",Profil([],[])))
              ((map__add 1 1
                (map__add 2 2 empty_int_mapping)),
               (map__add 3 1 empty_int_mapping))

    ( map__add (Oper("H",Profil([],[])))
              ((map__add 1 1 empty_int_mapping),
               (map__add 2 2 empty_int_mapping))

    ( map__add (Oper("P",Profil([],[])))
              ((map__add 1 1 empty_int_mapping),
               (map__add 2 1 empty_int_mapping))

              empty_param_mapping ) ) ) ) ;;
- : unit = ()
```

Nous construisons le terme à dériver :

```
#let t = Fonction("F",[Fonction("G",[Fonction("F",[Variable(Var_N_i("x",0));
                                                   Variable(Var_N_i("y",0));
                                                   Fonction("P",[Variable(Var_N_i("z",0))])
                                                  ]);
                                     Variable(Var_N_i("t",0))
                                    ]);
                        Fonction("G",[Variable(Var_N_i("x",0));
                                      Variable(Var_N_i("z",0))]);
                        Fonction("H",[Fonction("G",[Variable(Var_N_i("t",0));
                                                    Variable(Var_N_i("x",0))
                                                   ])
                                     ])
                        ]) ;;
```

```
t : terme =
 Fonction
  ("F",
   [Fonction
     ("G",
      [Fonction
        ("F",
         [Variable (Var_N_i ("x", 0)); Variable (Var_N_i ("y", 0));
          Fonction ("P", [Variable (Var_N_i ("z", 0))])]);
       Variable (Var_N_i ("t", 0))]);
    Fonction
     ("G", [Variable (Var_N_i ("x", 0)); Variable (Var_N_i ("z", 0))]);
    Fonction
     ("H",
      [Fonction
        ("G", [Variable (Var_N_i ("t", 0)); Variable (Var_N_i ("x", 0))])])])
```

Nous construisons l'arbre abstrait (les variables "fraîches" ont pour nom `variable` et un indice qui commence à 0) :

```
#let a = construit_arbre (t , Var_N_i("variable.",0)) ;;
a : arbre_affect =
 Noeud
  (1,
   [Noeud
     (5,
      [Noeud
        (6,
         [Feuille (Var_N_i ("x", 0)); Feuille (Var_N_i ("y", 0));
          Noeud
           (7, [Feuille (Var_N_i ("z", 0))],
            Soit
             (Var_N_i ("variable.0.1.1.", 1),
              Fonction ("P", [Variable (Var_N_i ("z", 0))])))],
         Soit
          (Var_N_i ("variable.0.1.", 1),
           Fonction
            ("F",
             [Variable (Var_N_i ("x", 0)); Variable (Var_N_i ("y", 0));
              Variable (Var_N_i ("variable.0.1.1.", 1))])));
       Feuille (Var_N_i ("t", 0))],
      Soit
       (Var_N_i ("variable.0.", 1),
        Fonction
         ("G",
          [Variable (Var_N_i ("variable.0.1.", 1));
           Variable (Var_N_i ("t", 0))])));
    Noeud
     (4, [Feuille (Var_N_i ("x", 0)); Feuille (Var_N_i ("z", 0))],
      Soit
       (Var_N_i ("variable.0.", 2),
        Fonction
         ("G", [Variable (Var_N_i ("x", 0)); Variable (Var_N_i ("z", 0))])));
    Noeud
     (2,
      [Noeud
        (3, [Feuille (Var_N_i ("t", 0)); Feuille (Var_N_i ("x", 0))],
```

```
        Soit
         (Var_N_i ("variable.0.3.", 1),
          Fonction
           ("G", [Variable (Var_N_i ("t", 0)); Variable (Var_N_i ("x", 0))]))))],
     Soit
      (Var_N_i ("variable.0.", 3),
       Fonction ("H", [Variable (Var_N_i ("variable.0.3.", 1))]))))],
  Soit
   (Var_N_i ("variable.", 0),
    Fonction
     ("F",
      [Variable (Var_N_i ("variable.0.", 1));
       Variable (Var_N_i ("variable.0.", 2));
       Variable (Var_N_i ("variable.0.", 3))])))
```

Nous affichons la valeur du terme et de l'arbre codé de manière plus lisible:

```
#printf__print(fmt_terme(t)) ;;
F(G(F(x0,y0,P(z0)),t0),G(x0,z0),H(G(t0,x0)))
- : unit = ()

#printf__print(fmt_arbre(a)) ;;
noeud 7: variable.0.1.1.1 := P(z0)
noeud 6: variable.0.1.1 := F(x0,y0,variable.0.1.1.1)
noeud 5: variable.0.1 := G(variable.0.1.1,t0)
noeud 4: variable.0.2 := G(x0,z0)
noeud 3: variable.0.3.1 := G(t0,x0)
noeud 2: variable.0.3 := H(variable.0.3.1)
noeud 1: variable.0 := F(variable.0.1,variable.0.2,variable.0.3)
- : unit = ()
```

Nous calculons pour le nœud racine les attributs `U` et `M` qui, pour chaque nœud, recensent les utilisations et modifications des variables faites dans son sous-arbre. Les arcs $\mathcal{R}_<$ sont calculés en même temps:

```
#let ( g_all_pre_util_map
     ,g_all_pre_mod_map
     ,g_l_ordres) =  var_util_mod a ;;
g_all_pre_util_map : (variable, int list) map__t = <abstr>
g_all_pre_mod_map : (variable, int list) map__t = <abstr>
g_l_ordres : ((int list * int list) * variable) list =
 ([[6], [4]), Var_N_i ("x", 0); ([5], [3]), Var_N_i ("t", 0);
  ([4], [7]), Var_N_i ("z", 0); ([3], [4]), Var_N_i ("x", 0)]
```

Nous calculons l'ensemble des arcs $\mathcal{R}_T$ de l'arbre abstrait, et avec l'ensemble des arcs $\mathcal{R}_<$ que nous venons de calculer, nous cherchons les cycles qui peuvent exister (les cycles trouvés sont indexés par les arcs de $\mathcal{R}_<$):

```
#let s_a = ordre_construct a ;;
s_a : (int list * int list) list =
 [[5], [1]; [4], [1]; [2], [1]; [6], [5]; [7], [6]; [3], [2]]

#let c_a = rechercher_cycles g_l_ordres s_a ;;
c_a : (int * int) list list list =
 [[[6, 4; 4, 7; 7, 6]]; [[5, 3; 3, 4; 4, 7; 7, 6; 6, 5]]; []; []]
```

Nous avons trouvé un cycle qui passe par l'arc $(6, 4)$. Une fois cet arc supprimé, un cycle passe par l'arc $(5, 3)$. Il n'y en a ensuite pas d'autres.

Nous calculons les coupures possibles de ces cycles, parmi les arcs de $\mathcal{R}_<$. Les variables sont supposées avoir le même poids, et la fonction passée en paramètre qui, entre deux listes de coupures renvoie celle dont le coût en duplications est le moindre, fait une simple comparaison de la longueur des deux listes :

```
#let j_a = coupe_cycles (fun x y ->
                         if ((list_length (hd x)) <= (list_length (hd y)))
                         then x else y)
                        g_l_ordres s_a ;;
j_a : (int * int) list = [4, 7]
```

Nous avons trouvé un arc : $(4, 7)$ qui coupe les deux cycles, et qui a donc été choisi comme coupure. Nous supprimons cet arc de l'ensemble des arcs de $\mathcal{R}_<$, et nous devrons ajouter la sauvegarde de sa variable conflictuelle : z en tête du texte engendré, et remplacer z par sa copie dans le nœud utilisateur : $\mathcal{N}_4$. Nous pouvons utiliser cette méthode si nous supposons qu'il n'existe pas plusieurs coupures de cycles qui ont une même variable conflictuelle. Dans le cas contraire, nous devons utiliser la méthode décrite à la section 4.6 page 99.

Nous calculons maintenant l'ordre dans lequel doit s'effectuer l'exécution des textes dérivés de chaque nœud dans le domaine concret, ici en choisissant un parcours en profondeur d'abord, et de droite à gauche pour les opérandes. Cet ordre doit respecter les ordres de $\mathcal{R}_T$ et de $\mathcal{R}_<$ moins les coupures de cycles. Nous pouvons exécuter le texte engendré pour $\mathcal{N}_7$ avant celui engendré pour $\mathcal{N}_4$. Nous affichons ensuite cette séquence de nœuds de manière plus lisible :

```
#let s = sequentiel g_l_ordres s_a j_a  a ;;
s : int list = [7; 6; 5; 3; 2; 4; 1]

#let s2 = map (fun no_noeud -> nommer_noeud no_noeud a) s;;
s2 : string list =
 ["noeud 7: variable.0.1.1.1 := P(z0) \n";
  "noeud 6: variable.0.1.1 := F(x0,y0,variable.0.1.1.1) \n";
  "noeud 5: variable.0.1 := G(variable.0.1.1,t0) \n";
  "noeud 3: variable.0.3.1 := G(t0,x0) \n";
  "noeud 2: variable.0.3 := H(variable.0.3.1) \n";
  "noeud 4: variable.0.2 := G(x0,z0) \n";
  "noeud 1: variable.0 := F(variable.0.1,variable.0.2,variable.0.3) \n"]

#map printf__print (rev s2) ;;
noeud 7: variable.0.1.1.1 := P(z0)
noeud 6: variable.0.1.1 := F(x0,y0,variable.0.1.1.1)
noeud 5: variable.0.1 := G(variable.0.1.1,t0)
noeud 3: variable.0.3.1 := G(t0,x0)
noeud 2: variable.0.3 := H(variable.0.3.1)
noeud 4: variable.0.2 := G(x0,z0)
noeud 1: variable.0 := F(variable.0.1,variable.0.2,variable.0.3)
- : unit list = [(); (); (); (); (); (); ()]
```

Seul le nœud $\mathcal{N}_5$ est exécuté par anticipation (avant le nœud $\mathcal{N}_3$, pour cause de conflit d'accès à t). Le code engendré pour $\mathcal{N}_5$, s'il était de forme fonctionnelle, ne

pourrait pas être paramètre effectif de F. Le texte Ada obtenu pour T est :

```
(⟦ duplication(z, z2);
   P(z);
   v1 := F(x, y, z);
   G(v1, t);
   G(t, x);
   G(x, z2);            ⟧, ⟨ F(H(t), x, v1) ⟩)
```

Conclusion

Bilan

Ce travail présente notre contribution à la promotion des spécifications algébriques pour la construction des programmes. Devant les exigences de plus en plus fortes de rapidité et de lisibilité des développements, et de sûreté et efficacité des programmes, les informaticiens travaillent à des niveaux d'abstraction plus élevés, et les outils informatiques doivent combler ce fossé entre ce que nous sommes capables de décrire, et ce que la machine sait faire. Les langages de programmation "modernes" manipulent des objets fonctionnels (Lisp, Scheme, CAML), des objets composites avec héritage (Smalltalk, C++, Ada 95) et les environnements de spécification se développent (Larch, Z, B, SpecWare).

Nous avons proposé une méthode qui permet de réaliser le développement au niveau des spécifications algébriques, et de dériver automatiquement ces spécifications dans un langage impératif en obtenant un code efficace. Le principe de décrire un ensemble de domaines de valeurs accompagné d'un ensemble d'opérations sur ces domaines de valeurs est celui des types abstraits. Modéliser un type abstrait par une algèbre est motivé par l'analogie entre un type abstrait et une algèbre multi-sorte sur une signature. L'interprétation des sortes correspond en effet aux domaines de valeurs, et l'interprétation des opérateurs correspond aux fonctions sur les domaines de valeurs. Nous dérivons séparément les sortes et opérateurs constructeurs en un type Ada, et les opérateurs définis en fonctions ou procédures Ada. Le code des fonctions et procédures est généré de manière à utiliser correctement les implémentations des spécifications importées de la bibliothèque.

Dans ce mémoire, nous avons défini formellement le concept principal de nos transformations : le lien d'implémentation ou lien de dérivation. Il relie dans le premier cas une spécification algébrique à une de ses implémentations impératives et, dans le deuxième cas, il paramètre la dérivation d'une spécification algébrique vers une implémentation impérative. Le lien de dérivation devient lien d'implémentation après que la dérivation ait été effectuée.

Nous avons énuméré l'ensemble des problèmes à résoudre dans la dérivation du fonctionnel vers l'impératif. Nous avons décrit ensuite les traitements préalables à la dérivation, puis le formalisme dans lequel nous exprimons le résultat impératif de la dérivation, avant sa traduction à la syntaxe Ada. Les traitements préalables consistent en une transformation de l'axiomatisation où plusieurs équations orientées définissent un opérateur, en une équation orientée équivalente où le filtrage est inclus.

Le principe de la dérivation a été ensuite décrit par une interprétation de chaque opération abstraite par une opération concrète. La mise en œuvre du lien d'implémentation pour effectuer cette interprétation a été alors montrée.

Nous avons défini une méthode de résolution des conflits d'accès aux variables. En effet l'interprétation des opérations abstraites par des opérations concrètes est correcte si ces dernières sont appliquées sur représentations intactes des opérandes des opérations abstraites, or les opérations du domaine concret peuvent modifier leurs paramètres. Des duplications sont alors nécessaires pour sauver les valeurs des variables rendues indisponibles car modifiées par une procédure. Nous ordonnons donc les calculs dans le domaine concret de façon à avoir un coût minimum en duplications.

La méthode que nous avons proposée est optimale si l'on ne considère pas les affectations finales aux paramètres formels de retour lorsqu'elles sont nécessaires, et le code obtenu reste le plus proche possible de l'ordre choisi pour l'évaluation (en largeur ou profondeur d'abord, de gauche à droite parmi les opérandes ou inversement, ou dans un ordre quelconque).

Nous avons montré comment dériver de manière systématique des sortes et leurs constructeurs (type abstrait algébrique) vers leurs représentations par des types Ada. Le lien d'implémentation qui correspond à chaque représentation est également généré. Les représentations standards sont définies dans une bibliothèque où diverses implémentations d'une unité algébrique sont possibles, avec des représentations de données qui peuvent être optimisées pour différents cas.

Nous avons défini un lien d'implémentation abstraite entre les sortes et les constructeurs de deux signatures, qui permet de réutiliser la représentation des données d'une spécification pour une autre, lorsqu'un lien existe entre la seconde et la première. Pour une spécification donnée, il est possible de rechercher automatiquement les spécifications de la bibliothèque qui ont un lien d'implémentation abstraite avec la première, et dont l'implémentation des sortes et des constructeurs est réutilisable. Cette méthode permet de généraliser de nombreuses optimisations.

Nous avons proposé des variantes de la méthode de dérivation des corps de programmes :

- Chaque sous-terme du terme à dériver hérite d'une destination de l'opérateur dont il est opérande : lorsque cet opérateur possède une destination pour son résultat, et qu'il est implémenté par une procédure dont le paramètre est en entrée-sortie à la position correspondant à celle de l'opérande. Nous avons ainsi défini des flots de données par une succession de calculs pour lesquels une même variable est à la fois opérande et destination. Nous avons donc économisé des affectations finales aux paramètres résultat à chaque fois que cela était possible.

- Le terme algébrique est transformé avant sa dérivation, en remplaçant par des mutations, les constructions d'un terme, presque identique à un autre terme, à partir de composants du second. Les conditions sont : le changement de seules quelques sous-structures d'un terme, avec pour destination du résultat la même variable que celle qui contient le terme d'origine.

Nous avons présenté un prototype de l'outil de dérivation, réalisé en CAML. À

partir d'un arbre abstrait, il résout les conflits d'accès aux variables en ordonnant les calculs de manière à minimiser le coût en duplications de données.

Perspectives

La méthode de dérivation que nous avons présentée et le formalisme que nous avons défini se prêtent facilement à des évolutions :

Extension de l'aspect "modification partielle de données"

Certains termes peuvent être avantageusement utilisés comme des adresses : une application de sélecteurs du domaine abstrait est souvent implémentée par un accès à des composants, qui est une expression affectable.

Dans la dérivation, telle qu'elle a été décrite, un constructeur ou un opérateur défini n'est jamais considéré comme ayant pour résultat une adresse dans le domaine concret. En effet le prédicat définissant une expression-gauche est défini de la manière suivante :

$$LeftExpr(x) \Leftarrow Var(x) \ \wedge \ \neg Param_{In}(x)$$

La partie fonctionnelle du code impératif dérivé d'un terme n'est jamais utilisée comme paramètre effectif modifié par une procédure, si ce code n'est pas une variable ou un paramètre formel en mode `out` ou `in out` de l'opérateur d'intérêt. Cela est vrai même si le code est dérivé en une composition de sélecteurs du domaine concret (déréférencement, accès à un champ, etc). L'affectation à une variable, du résultat d'une sélection d'une autre variable fait une duplication de la valeur sélectée.

Exemple 7.2: Pour une sorte *B-Arbre* ayant 2 sélecteurs $filsGauche$ et $filsDroit$, et un opérateur défini *balancer* dont les liens d'implémentation sont :

$$\mathcal{E}(filsGauche) = [x]\{\texttt{x.gauche}\},\ In(filsGauche) = <\pi_1>,$$
$$Out(filsGauche) = <\phi>$$
$$\mathcal{E}(filsDroit) = [x]\{\texttt{x.droite}\},\ In(filsDroit) = <\pi_1>,\ Out(filsDroit) = <\phi>$$
$$\mathcal{E}(balancer) = [x]\{\texttt{BALANCER(x)}\},\ In(balancer) = <\pi_1>,\ Out(balancer) = <\pi_1>$$

Soit un terme $t = balancer(filsGauche(a))$ à dériver, dont a est une variable, la dérivation est :

$$\Delta^{\#}(balancer(filsGauche(a)) = ([\![v_1 := a.gauche;\ BALANCER(v_1)]\!], \langle\!|v_1|\!\rangle)$$

■

L'extension consisterait à autoriser des termes dérivés en expression-gauche, en position de paramètre effectif modifié. Par exemple, nous pourrions permettre la dérivation :

$$\Delta^{\#}(balancer(filsGauche(a)) = ([\![BALANCER(a.gauche)]\!], \langle\!|a.gauche|\!\rangle)$$

en ayant obtenu $\Delta^{\#}(filsGauche(a)) = ([\![\ ]\!], \langle\!|a.gauche|\!\rangle)$.

La propriété "l'opérateur a pour résultat une expression-gauche" doit donc être rangée avec le lien de dérivation de cet opérateur.

La règle de calcul des attributs d'utilisation et modification `U` et `M` de chaque nœud doit alors être modifiée en conséquence pour :

- soit connaître parmi les sous-termes du terme dérivé en une adresse, la variable (partiellement) modifiée,
- soit, en limitant aux sélecteurs du domaine abstrait les opérateurs dont l'implémentation retourne une adresse, désigner par une composition de sélecteurs abstraits, la localisation de la modification.

Dans le premier cas, si g est un opérateur dont l'application dans le domaine concret donne une adresse, l'opérande de *op* qui contient la structure désignée par l'adresse, est une information qui doit également être rangée dans le lien d'implémentation de *op*.

Localisations plus fines des conflits

Nous avons considéré l'utilisation et la modification des variables telle qu'elle est définie par l'interface Ada des programmes. Or un paramètre en entrée-sortie sous-entend une *modification* de la variable (ou expression-gauche) passée en paramètre effectif, et donc que seule une sous-structure change.

Une recopie de sauvegarde n'est pas toujours nécessaire : les fonctions ou procédures qui utilisent le même paramètre effectif, après sa modification ne requièrent éventuellement qu'une partie qui n'a pas été modifiée. Nous pouvons alors ne pas avoir de conflit lorsque la partie modifiée d'une variable par le texte dérivé d'un terme n'est qu'une *sélection* du terme représenté par le contenu de la variable, et que la partie utilisée ultérieurement en est une sélection sans sous-terme commun avec la précédente.

Il est avantageux de décrire les sous-termes utilisés et modifiés par les implémentations des opérateurs, par des compositions de sélecteurs du domaine abstrait ($select_i^{constructeur}$). En effet, les informations que l'on peut calculer portent sur des données pertinentes, sans confusion avec ce qui appartient au codage.

Etant donné que ces informations ne peuvent pas être déterminées à partir de l'implémentation (spécification ou code Ada), elles doivent être décrites dans le lien d'implémentation. Elles peuvent être calculées pour les opérateurs que nous dérivons en fonction des utilisations et modifications des opérateurs du terme dérivé pour obtenir le corps de programme. Ce calcul est un point-fixe lorsqu'il s'agit d'un opérateur récursif. Une autre possibilité est de fixer pour chaque opérateur dérivé, la localisation des utilisations et modifications de ses opérandes. Un algorithme de dérivation proche de celui décrit au chapitre 4 résout les conflits, sans faire de duplication inutile, et en respectant les paramètres comme il a été précisé.

Extension à d'autres langages

La réalisation, comme l'approche présentées dans ce mémoire, sont suffisamment générales pour être appliquées à d'autres langages de spécification : nous partons d'un arbre abstrait, c'est-à-dire d'un code intermédiaire où il s'agit de réécrire un terme. L'application à un autre langage de spécification serait facilitée par le travail qui a été effectué sur la normalisation d'une plate-forme commune aux langages algébriques [Sal92a, Sal92b].

Quant aux langages impératifs cibles, certains langages de programmation traditionnels, comme le langage C, ne permettent pas de déduire les modifications de paramètres effectifs à partir d'une spécification concrète ou même du profil des fonctions : toutes les modifications sont des effets-de-bord (il s'agit du cas où nous intégrons dans la bibliothèque des programmes non générés automatiquement, dans le cas de la dérivation, un fichier interface peut donner cette information). Nous devons alors stocker ces informations dans le lien d'implémentation de chaque opérateur. Il est alors possible, comme il est décrit pour la localisation fine des conflits, soit de calculer les utilisations et modifications de l'opérateur d'intérêt par un calcul de point-fixe, soit de lui fixer ses propriétés (d'utilisation et modification), et de les faire respecter par l'algorithme de dérivation.

Globalisation de paramètres

Nous pouvons transformer en variables globales certains paramètres formels de l'opérateur d'intérêt. Soit un opérateur d'intérêt *op* implémenté par une fonction ou procédure `OP`, et dont le terme T est dérivé en corps de procédure de *op*. Le paramètre formel x de l'opérateur d'intérêt *op* est candidat à une transformation en une variable globale `X` si la dérivation de T vérifie :

> Soit un sous-terme t de T qui utilise x. Le texte impératif dérivé de t peut être exécuté *après* la dérivation d'un appel récursif de *op* uniquement si :
>
> - en mode `In` : l'opérande de *op* à la position de x est x,
> - en mode `In-Out` : l'opérande de *op* à la position de x est x, et la modification de x par *op* est localisée à une sous-structure disjointe de celle utilisée par le texte dérivé de t.

En effet, dans le domaine concret, un appel récursif de *op* implique une affectation à la variable globale `X` de la valeur de l'opérande à la position de x.

La méthode de dérivation que nous proposons nous permet de privilégier cet ordre d'exécution des textes dérivés, et de faire des duplications de sauvegarde de `X` pour son utilisation après un appel récursif à `OP`, si cet ordre est incompatible avec d'autres ordres d'exécution (c'est-à-dire si elle crée des cycles).

L'avantage de transformer en variables globales est d'éviter au compilateur de ranger sur une pile le paramètre utilisé. Nous pouvons trouver des travaux sur ce sujet dans [Gou95] où la propriété est déduite d'analyses de traces, et dans [Fra91] où elle liée à l'utilisation des "continuations" dans les langages fonctionnels.

Traitement des effets de bord

Lorsqu'un ordre de réduction est précisé explicitement à l'utilisateur (appelé "ordre apparent") nous pouvons ajouter des ordres d'exécution entre noeuds qui codent les sous-termes de manière à respecter la condition suivante : le texte dérivé du sous-terme qui fait un effet de bord doit être exécuté avant le texte dérivé d'un sous-terme sur lequel l'effet de bord a une conséquence. Soit un graphe qui code l'ordre dans lequel s'effectuent les calculs, et les nœuds de ce graphe qui codent les termes algébriques, ce peut être par exemple : seuls sont pris en compte pour la dérivation les ordres apparents

pour l'utilisateur : "nœud $\mathcal{N}$ avant nœud $\mathcal{N}'$" tels que les textes dérivés pour appliquer les opérateurs des noeuds $\mathcal{N}$ et $\mathcal{N}'$ effectuent une entrée ou sortie vers l'écran ou le clavier.

Dans notre algorithme de dérivation, les utilisations et modifications de variables effectuées par l'appel et l'exécution d'une procédure respectent les modes des paramètres formels de son interface Ada, et sont calculées à partir du lien d'implémentation de l'opérateur qui est implémenté par cette procédure. Pour les effets de bord, nous devons stocker dans le lien d'implémentation l'information qu'une fonction ou procédure fait des utilisations ou modifications du fichier "écran-clavier", de fichiers quelconques, de canaux de communication, etc.

Parallélisation

Le graphe orienté acyclique obtenu par notre méthode après la résolution des conflits d'accès aux variables code le calcul du terme T en partie droite de l'équation qui définit l'opérateur d'intérêt. Les arcs de ce graphe codent l'ordre des calculs des sous-termes de T à respecter dans le domaine concret. Chaque nœud n'ayant aucun arc entrant n'a pas de calcul "prédécesseur". L'ensemble de ces arcs peut donc être exécuté en parallèle. Les arcs qui partent de ces nœuds peuvent être supprimés, et nous obtenons un nouvel ensemble de nœuds qui peuvent être exécutés en parallèle. Nous pouvons continuer ainsi jusqu'à exécuter le nœud du graphe qui code T.

Bibliographie

[Abr91] J.R. Abrial. The B-method for large software specification, design and coding. *VDM-91*, 2, 1991.

[AFQ86] F. Alexandre, J.P. Finance, and A. Quéré. SPES: Un Système de Transformation de Programmes Logiques. In *Actes du colloque sur la programmation en logique (CNET, Tregastel)*, 1986.

[ALS87] ALSYS. *Manuel de référence du langage de programmation ADA*, 1987.

[BA90] Didier Bert and Cyril Autant. Dérivation assistée de programmes. Rapport de projet PROPOS – MRT 88.S.1104, LIFIA 2 – IMAG, Décembre 1990.

[Bac78] John W. Backus. Can Programming be liberated from the Von Neumann style? Rapport numero RJ 2234, IBM US Research Center. Yorktown, San Jose, Almaden (US), May 1978.

[BB92] Gilles Bernot and Michel Bidoit. Proving the correctness of algebraically specified software: modularity and observability issues. In *Proc. of the 2nd International Conference on Algebraic Methodology and Software Technology (AMAST, Iowa City, U.S.A., May 1991)*, Workshops in Computing, pages 216–242. Springer-Verlag, 1992. Invited paper.

[BD77] R.M Burstall and J. Darlington. A transformation system for developping recursive programs. *Journal of the ACM*, 24(1), January 1977.

[BE86] D. Bert and R. Echahed. Design and implementation of a generic, logic and functional programming language. In *Proceedings of the European Symposium on Programming; Saarbrücken*, number 213 in LNCS, pages 119–132, 1986.

[Bee85] Michael J. Beeson. *Foundations of Constructive Mathematics, Metamathematical Studies*, volume 006, chapter XXIII, page 466. Springer Verlag – Berlin ; New York ; Tokyo, 1985.

[Ber73] Claude Berge. *Graphes et hypergraphes.* Dunod, $2^{ème}$ edition, 1973.

[Ber82] Didier Bert. Software components constructions. In D. Néel, editor, *Tool and Notions for Program Construction*, pages 347–376. Cambridge University Press, 1982.

[Ber86] Gilles Bernot. Correctness Proofs for Abstract Interpretations. In New York/Berlin Springer Verlag, editor, *Proceedings, 3rd STACS, Vol. 210*, number 210 in LNCS, pages 236–251. Academic Press, 1982., January 1986.

[BG80] R.M. Burstall and J.A Goguen. The semantic of CLEAR, a specification language. In *Advanced Course on Abstract Software Specifications*, number 86 in LNCS, pages 292–332, 1980.

[Bid89] Michel Bidoit. *PLUSS, un langage pour le développement de spécifications algébriques modulaires.* PhD thesis, Université Paris-Sud, 1989. Thèse d'Etat.

[BLY93] Valdis Berzins, Luqi, and Amiram Yehudai. Using Transformations in Specification-Based Prototyping. In IEEE Transactions on Software Engeneering, editor, *Proceedings, Vol. 19, No. 5*, pages 434–452, May 1993.

[BS91] Didier Bert and Sadik Sebbar. Synthesizing abstract data type representation in the DEVA meta-calculus. In B. Möller, editor, *IFIP TC2 Working Conference on Constructing Programs from Specifications, Pacific Grove (USA)*, pages 427–450, Amsterdam, May 1991. North-Holland.

[CABC86] (R.L.) Constable, (S.F.) Allen, (Hank M.) Bromley, and (W.R.) Cleaveland. *Implementing mathematics with the Nuprl proof development system.* Prentice-Hall, 1986.

[CH87] Thierry Coquand and Gérard Huet. Concepts mathématiques et informatiques formalisés dans le Calcul des Constructions. In The Paris Logic Group, editor, *Logic Colloquium'85. North-Holland*, 1987. Rapport numero: RR 0515, 04-1986.

[CK90] Christine Choppy and S. Kaplan. Mixing abstract and concretes modules: specification, development and prototyping. In IEEE Transactions on Software Engeneering, editor, *Proceedings of the 12th International Conference on Software Ingeniering*, pages 173–184, 1990.

[Coo85] Keith Cooper. Analysing Aliases of Reference Formal Parameters. In *Conf. Rec. Twelfth ACM Symposium on Principle of Programming Languages*, pages 281–290, 1985.

[Cou77] Patrick and Radhia Cousot. Static verification of dynamic type properties of generalized type unions. In NC Raleigh, editor, *ACM Symposium on Language Design for Reliable Software*, pages 77–94, 1977.

[Cou92] Patrick and Radhia Cousot. Abstract Interpretation Framework. Report 05-1992 revised version, Ecole Polytechnique, Palaiseau, May 1992.

[Dar75] John Darlington. Application of program transformation to program synthesis. In *Proceedings of International Symposium on Proving and Improving Programs. Arc et Senans, France*, 1975.

[Dar83] John Darlington. Program transformation. In Turner (D.A.) Darlington (J.), Henderson (P.), editor, *Functional programming and its applications : an advanced course*, pages 193–215. Camdridge University Press, 1983.

[Deu89] Alain Deutch. On determining lifetime and aliasing of dynamically allocated data and higher-order functionnal specifications. In Christian Queinnec, editor, *ICSLA (Interpretation,Compilation et Semantique des Langages Applicatifs)*. ACM, juillet-decembre 1989.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall series in automatic computation. Edsger W. Dijkstra. - Englewood Cliffs, NJ : Prentice-Hall, 1976.

[DP90] (B.A.) Davey and (H.A) Priestley. *Introduction to lattices and order*. Cambridge University Press, 1990.

[DS93] R. Darimont and J. Souquières. A Development Model : Application to Z Specifications. In N. Prakash, C. Rolland, and B. Pernici, editors, *Proceedings IFIP WG 8.1 Conference on Information System Development Process*, Come (Italy), September 1993.

[EKMP82] H. Ehrig, H.-J. Kreowski, B. Mahr, and P. Padawitz. Algebraic implementation of abstract data types. *TCS (Theoretical Computer Science)*, 20:209–263, 1982.

[EKP80] H. Ehrig, H. Kreowski, and P. Padawitz. Algebraic implementation of abstract data types: concept, syntax, semantics and correctness. In *ICALP*, number 85 in LNCS, 1980.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of algebraic specification 1. Equations and initial semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.

[EM90] H. Ehrig and B. Mahr. *Fundamentals of algebraic specification 2. Module Specifications and Constraints*, volume 21 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1990.

[FC89] Jordi Farres-Casals. Proving correctness of constructor implementations. Rapport numero : Ecs-lfcs-89-072, University of Edinburgh. Edimbourg (GB), January 1989.

[Fin79] J-P Finance. *Etude de la Construction des Programmes: méthodes et langages de Spécification et de Résolution de Problèmes*. PhD thesis, Université de Nancy I, Octobre 1979.

[Fra91] Pascal Fradet. Syntactic Detection of Single-Threading using Continuations. In *Proceedings of FPCA'91*, number 523 in LNCS, pages 241–258, 1991.

[Gab91] Robert Gabriel. Program transformation expressed in the DEVA metacalculus. In B. Möller, editor, *IFIP TC2 Working Conference on Constructing Programs from Specifications, Pacific Grove (USA)*, Amsterdam, May 1991. North-Holland.

[Gau84] Marie-Claude Gaudel. A first introduction to PLUSS. Technical report, LRI, Université Paris-Sud, Orsay, 1984.

[GH93] John V. Guttag and James J. Horning, editors. *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer-Verlag, 1993. With Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing.

[GL86] David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *ACM Conference on Lisp and Functional Programming*, pages 28–38, 1986.

[Gor79] Michael J.C. Gordon. *The denotational description of programming languages.* Springer Verlag, 1979.

[Gou95] Valérie Gouranton. Une analyse de globalisation de programmes fonctionnels basée sur une sémantique naturelle. In *Proceedings of Journées du GDR de Programmation*, novembre 1995.

[Gri82] David Gries. *The Science of Programming.* Texts and monographs in computer science. David Gries. - New York ; Heidelberg ; Berlin : Springer, 1982.

[Gri92] David Gries. Lectures on Data Refinement. In *Programming and Mathematical Method, International Summer School Martobendorf 1990*, volume 88. NATO ASI Series, Series F: Computer and Systems Sciences, 1992.

[GTW78] J. A. Goguen, J. W. Thatcher, and E. G. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In R. T. Yeh, editor, *Current Trends in Programming Methodology*, volume 4: Data Structuring, pages 80–149. Prentice-Hall, 1978.

[GTWW75] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Abstract data types as initial algebras and the correctness of data representation. In *Computer Graphics, Pattern Recognition and Data Structure*, pages 89–93, 1975.

[GTWW77] J. A. Goguen, J. W. Tatcher, E. G. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebras. In *Programming and Mathematical Method, International Summer School Martobendorf 1990*, volume 24, pages 68–95. Journal of ACM, 1977.

[Gut75] John V. Guttag. *The Specification and Application to Programming of Abstract Data Types.* PhD thesis, University of Toronto, Department of Computer Science, 1975.

[Gut77] John V. Guttag. Abstract data types and the development of data structures. In *Communication of the ACM*, volume 6, pages 396–404, 1977.

[Hey66] A. Heyting. Intuitionism, an introduction. In *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1966.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. In *Communications of the ACM*, volume 12, pages 576–583, October 1969.

[Hoa72] C.A.R. Hoare. Proofs of correctness of data representations. In Springer Verlag, editor, *ACTA INFORMATICA*, volume 1, pages 271–281, 1972.

[How69] W.A. Howard. The formulae-as-types notion of construction. In J.P. Seldin and J.R. Hindley, editors, *to H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, 1980, 1969. Unpublished 1969 manuscript.

[JLM+94] D. T. Jordan, C. J. Locke, J. A. McDermid, C. E. Parker, B. A. P. Sharp, and I. Toyn. Literate Formal Development of Ada from Z for Safety Critical Applications. In *SAFECOMP94*, 1994.

[Jon86] Cliff B. Jones. *Systematic software development using VDM*. Prentice-Hall international series in computer science. Cliff B. Jones. - Englewood Cliffs, NJ ; London ; Mexico : Prentice-Hall, 1986.

[Kah86] Stefan Kahrs. From Constructive Specifications to Algorithmic Specifications. PROSPECTRA report M. 3.1.S-SN-1.1, FB3 Mathematik und Informatik, Universität Bremen, September 1986.

[KB86] Bernd Krieg-Brückner. Systematic Transformation of Interface Specification. PROSPECTRA report M. 1.1.S1-R-3.1, FB3 Mathematik und Informatik, Universität Bremen, April 1986.

[KGJM85] K.Futatsugi, J.A Goguen, J.P Jouannaud, and J. Meseguer. Principles of OBJ2. In *Proceedings of POPL, Principle of Programming Languages*, pages 52–56, 1985.

[Kle52] S.C. Kleene. *Introduction to metamathematics*. North-Holland, bibliotheca mathematica edition, 1952.

[Kna93] Teodor Knapik. Specifications with Observable Formulae and Observational Satisfaction Relation. In Michel Bidoit and Christine Choppy, editors, *8th workshop on specification of abstract data types, joint with the 3rd COMPASS workshop, Dourdan, France, August 1991*, number 655 in LNCS, pages 271–291. Springer Verlag, 1993.

[KP87] J.L. Krivine and M. Parigot. Programming with Proofs. In *6th symposium on Computation Theory, Wendish-Rietz, Germany*, 1987.

[KS86] U. Kastens and M. Schmidt. Lifetime Analysis for Procedure Parameters. In G. Goos and J. Hartmanis, editors, *European Symposium on Programming*, number 213 in LNCS, pages 53–69. Springer Verlag, March 1986.

[Lin93] Huimin Lin. Procedural Implementation of Algebraic Specification. In *ACM Transactions on Programming Languages and Systems, Vol. 15, No. 5*, pages 876–895, November 1993.

[Los91] Francisca Losavio. *Dérivation de Programmes Ada comportant des Traitements d'Exceptions à partir de Spécifications Algébriques de Types de Données.* PhD thesis, Université de Paris XI Orsay, Novembre 1991.

[LZ74] Barbara H. Liskov and Stephen N. Zilles. Programming with abstract data types. Report CSG-M-099, Massachussetts Institute of Technology. Cambridge (MA US), March 1974.

[MGRV92] Carroll Morgan, Paul Gardiner, Ken Robinson, and Trevor Vickers. *On the Refinement Calculus.* Springer Verlag Berlin Heidelberg New-York, 1992. C. Morgan and T. Vickers editors.

[ML80] Per Martin-Löf. Intuitionistic type theory. In *Studies in proof theory: lecture notes*, volume 1. Bibliopolis, 1980.

[Mor90] Carroll Morgan. *Programming from specifications.* Prentice Hall international series in computer science, 1990.

[MR74] E. Morel and C. Renvoise. *Etude et Réalisation d'un Optimiseur Global.* PhD thesis, Université de Paris VI, Juin 1974.

[Mul89] Jean-Michel Muller. *Arithmétique des ordinateurs : opérateurs et fonctions élémentaires*, pages 34–40. Masson (Paris), 1989.

[MVS85] T.S.E Maibaum, Paulo A.S. Veloso, and M.R. Sadler. A theory of abstract data types for program development: bridging the gap? In *Proceedings of TAPSOFT, Berlin, Formal Methods and Software Development*, number 186 in LNCS, pages 214–230, 1985.

[MW71] Z. Manna and R. Waldinger. Toward automatic program synthesis. *Communications of the ACM*, 14(3), march 1971.

[MW81] Z. Manna and R. Waldinger. *Deductive Synthesis of the unification algorithm*, volume 1 of *Science of Computer Programming*, pages 5–48. North-Holland Publishing Company, 1981.

[Ore93] Fernando Orejas. Implementation and Behavioural Equivalence: A Survey. In Michel Bidoit and Christine Choppy, editors, *8th workshop on specification of abstract data types, joint with the 3rd COMPASS workshop, Dourdan, France, August 1991*, number 655 in LNCS, pages 93–125. Springer Verlag, 1993.

[Ori96] Catherine Oriat. *Etude des spécifications modulaires: constructions de colimites finies, diagrammes, isomorphismes.* Doctorat d'université, inpg, grenoble, INPG, Grenoble, 1996.

[Pep91] Peter Pepper. *Transforming Algebraic Specifications – Lessons Learnt From an Example*, pages 399–425. Elsevier Science Publishers B.V. (North Holland), 1991. B. Möller editor.

[PM89] C. Paulin-Mohring. Extracting F_Ω's programs from proofs in the Calculus of Constructions. In ACM – Association for Computing Machinery, editor, *Sixteenth annual ACM Symposium on Principles of Programming Languages, Austin*, 1989.

[Pot88] Marie-Laure Potet. *Preuves et Stratégies pour la Synthèse déductive de Programmes*. PhD thesis, Institut National Polytechnique de Grenoble, Juin 1988.

[PP88] A. Pettorossi and M. Proietti. The Automatic Construction of Logic Programs. Technical report, Istituto di analisi dei sistemi ed informatica. Rome (IT), 1988.

[PS81] Helmut Partsch and R. Steinbruggen. A comprehensive survey on program transformation systems. Report TUM-INFO-8108, Technische Universität München. Munich (DE), July 1981.

[Sak84a] Michel Sakarovitch. *Optimisation combinatoire : méthodes mathématiques et algorithmes : graphes et programmation linéaire*, volume 031. Hermann, 1984.

[Sak84b] Michel Sakarovitch. *Optimisation combinatoire : méthodes mathématiques et algorithmes : programmation discrète*, volume 032. Hermann, 1984.

[Sal92a] Salsa. Définition de la forme interne des spécifications algébriques dans la structure d'accueil SALSA. Technical Report R. G. 03-92, GRECO de Programmation, CNRS, Bordeaux, 1992.

[Sal92b] Salsa. Le projet SALSA : Structure d'AccueiL pour les Spécifications Algébriques. In *Compte-Rendu des Journées GROPLAN - GDR Programmation et Outils de l'Intelligence Artificielle, 18-20 mars 1992, Nancy*, pages 157–168, 1992.

[San90] Donald Sannella. Formal program development in extended ML for the working programmer. LFCS report series, CSR-321-89, University of Edinburgh, January 1990.

[Sch88] Philippe Schnoebelen. Refined compilation of pattern-matching for functional languages. Rapport numero : RR 715-I-71 LIFIA, IMAG, Laboratoire d'Informatique et de Mathématiques Appliquées de Grenoble, Université scientifique et médicale, Grenoble 1, Avril 1988.

[SD95] J. Souquières and R. Darimont. La description du développement de spécifications. *Technique et Science Informatiques*, 14(9):1073–1096, novembre 1995.

[Sek88] H. Seki. Unfold/Fold Transformations of Stratified Programs Programs. Technical memorandum TM-0240, ICOT, 1988.

[SF86] H. Seki and K. Furukawa. Notes on Transformation Techniques for Generate and Test Logic Programs. Technical memorandum TM-0240, ICOT, 1986.

[Sig95] Mihaela Sighireanu. Méthode de Représentation des Types Abstraits Algébriques en vue de la Vérification de Protocoles. Mémoire de DEA, ENSIMAG, Institut National Polytechnique de Grenoble, Juin 1995.

[Sou93] J. Souquières. Aides au Développement de Spécifications. Rapport de recherche 93-T-013, Centre de recherche en Informatique de Nancy, Janvier 1993.

[ST87] Donald Sannella and Andrzej Tarlecki. Toward formal developpement of programs from algebraic specifications: implementation revisited. In Springer, editor, *TAPSOFT'87*, number 249 in LNCS, pages 96–110, 1987.

[Tan94] Yang Meng Tan. Interface language for supporting programming styles. In *POPL Workshop on Interface Definition Languages*, Portland, Oregon, Januar 1994.

[Toe73] A.S. Toelstra. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis.* Springer Verlag, LNM 344 edition, 1973.

[Tur92] Philippe Turlier. La compilation des types abstraits du langage LOTOS. Mémoire Ingénieur CNAM, Conservatoire National des Arts et Métiers, Décembre 1992.

[TvD88] A.S. Toelstra and D. van Dalen. Constructivism in Mathematics, an introduction. In *Studies in Logic and the foundation of Mathematics*, volume 121 and 123. North-Holland, 1988.

[Van94] Mark T. Vandevoorde. Exploiting specifications to improve program performance. Technical Report LCS/TR–598, MIT, February 1994. Ph. D. Thesis, Department of Electrical Engineering and Computer Science.

[Wei80] William E. Weihl. Interprocedural Data Flow Analysis in the Presence of Pointers, Procedure Variables, and Label Variables. In *7th ACM Symposium on Principle of Programming Languages*, pages 83–94, 1980.

[Wir71] Niklaus Wirth. Program Development by Stepwise Refinement. In P.Wegner, editor, *Communications of the ACM, Number 4*, volume 14, pages 221–227, April 1971.

[Wir90] M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 13, pages 677–788. Elsevier Science Publishers B.V., 1990.

[Wir94] M. Wirsing. Algebraic specification languages: An overview. In *Recent Trends in Data Type Specification, 10^{th} Workshop on Specification of Abstract Data Types*, number 906 in LNCS, pages 81–115, 1994.

[Zam95] Nancy Zambrano. *Une méthode de dérivation de programmes impératifs à partir de spécifications algébrico-opérationnelles.* PhD thesis, Université de Paris XI Orsay, Septembre 1995.