INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Mises à jour destructives dans les grammaires attribuées

Etienne DURIS , Didier PARIGOT , Martin JOURDAN

N° 2686

Octobre 1995

PROGRAMME 2

Rapport de recherche

ISSN 0249-6399

Mises à jour destructives dans les grammaires attribuées

Etienne Duris , Didier Parigot , Martin Jourdan

Programme 2 — Calcul symbolique, programmation et génie logiciel
Projet Charme

Rapport de recherche n° 2686 — Octobre 1995 — 9 pages

Résumé : Dans le domaine des langages fonctionnels, il existe un ensemble de travaux sur les problèmes liés à la récupération de mémoire au moment de la compilation. Dans le domaine des grammaires attribuées, le problème de l'occupation mémoire a donné lieu à des recherches sur la notion de durée de vie des attributs. Nous présentons une méthode permettant, dans le contexte des grammaires attribuées, de remplacer des appels à des fonctions par leurs équivalents destructifs. Si l'on considère le programme fonctionnel équivalent à une grammaire attribuée donnée, notre technique peut être vue comme une cométhode statique de récupération de mémoire à la compilation sur celui-ci. Cette méthode utilise les résultats obtenus sur la notion de durée de vie des attributs. L'une de ses particularités est qu'elle n'utilise que des méthodes d'analyse statique classiques des grammaires attribuées.

Mots-clé : Grammaires attribuées, mises à jour destructives, analyse de durée de vie.

(Abstract: pto)

. Ce travail a été partiellement financé par le projet ESPRIT #5399 "Compare".

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : (33 1) 39 63 55 11 – Télécopie : (33 1) 39 63 53 30

Update-in-place Transformations on Attribute Grammars

Abstract: In the functional language domain, there exist several works on the problem of garbage collection at compile time. In the context of Attribute Grammars, the memory optimisation problem spawned a number of works on the notion of attribute lifetime. We present a method which, in this context, allows to replace some function calls with their destructive counterpart. If we consider the functional program which is equivalent to a given attribute grammar, our technique can be seen as a static method for its update-in-place transformation. This method uses the results obtained on the attribute lifetime notion. One of its features is that it uses only classical static analysis methods for attribute grammars.

Key-words: Attribute grammars, update-in-place transformation, lifetime analysis.

1 Introduction

Les grammaires attribuées [DJL88] sont connues depuis un certain temps comme une méthode déclarative et structurée de spécification d'un calcul sur une structure arborescente. Plus précisément, on décompose les calculs sur chaque composante (constructeur) de la structure. Sur chacune d'entre elles, on spécifie localement le calcul à l'aide de définitions d'attributs, communément appelées règles sémantiques. Cette méthode est dite déclarative au sens où l'écrivain ne se soucie pas de l'ordre des calculs. D'autre part, la théorie des grammaires attribuées n'impose qu'une seule restriction sur le langage de codage des règles sémantiques : il faut qu'il soit sans effets de bord.

De nombreux travaux ont traité du problème de la génération des évaluateurs d'attributs (qui sont les programmes qui exécutent le calcul décrit par la spécification). L'un des problèmes liés à l'exécution de ces évaluateurs est leur coût en espace mémoire. En effet, les premières méthodes d'évaluation stockaient les valeurs des attributs sur l'arbre d'entrée et ne cherchaient pas à récupérer l'espace mémoire occupé par une valeur d'attribut après que celle-ci soit devenue inutile ; compte tenu du nombre important de nœuds important dans un arbre et du nombre d'instances d'attributs par nœud, lui aussi important, la taille mémoire totale devenait prohibitive. Aussi, un ensemble de recherches a visé à optimiser l'occupation mémoire [Jul89] [JP90] [JPJ+90]. Ces travaux ont cherché à déterminer de façon statique la durée de vie des instances d'attributs (de leur définition à leur dernière utilisation), ce qui permet de décider de l'implantation d'un attribut en variable globale ou en pile plutôt que sur l'arbre.

Etant donné cette analyse, certaines définitions d'attributs peuvent être codées dans l'évaluateur sous la forme $v := f(v)$, où f est une fonction applicative et v une variable globale. Dans ce cas, nous nous sommes posé le problème du remplacement de f par son équivalent destructif ($f_d(v)$). Par exemple, si f est la fonction d'insertion dans un arbre binaire de recherche v, son remplacement par la fonction destructive associée (qui réutilise sur place la structure de v) engendre un gain d'espace mémoire.

Pour répondre à cet objectif, nous avons d'une part à définir la transformation d'une fonction en son équivalent destructif et d'autre part à déterminer statiquement la notion de *destructibilité* de la variable globale v. Celle-ci ne doit pas être partagée par d'autres objets. Cette notion de destructibilité nous assure de la validité (correction) de ce remplacement.

De plus, à une grammaire attribuée donnée peut être associé un programme fonctionnel équivalent [1][Joh87]. Ainsi, la transformation d'une grammaire attribuée qui consiste à remplacer des appels de fonctions par leurs versions destructives peut être vue comme la transformation du programme fonctionnel équivalent, dont l'objectif est de faire de la récupération mémoire en place (*compile-time garbage collection*).

Il est clair que notre problème nous a amené à étudier les travaux analogues existant en programmation fonctionnelle. Ces travaux introduisent des concepts similaires comme la durée de vie des variables et leur partage. En revanche, les mécanismes mis en œuvre pour calculer ces propriétés de durée de vie et de partage sont assez différents des méthodes de calcul de durée de vie dans les grammaires attribuées. En programmation fonctionnelle, ces mécanismes sont en général le comptage de références, les chemins d'utilisation-définition et/ou l'interprétation abstraite [Hed88] [Deu94].

Nous n'avons pas utilisé directement les mêmes méthodes qu'en programmation fonctionnelle pour répondre à notre objectif, puisque d'une part il existe déjà un résultat sur la durée de vie des attributs et, d'autre part, nous disposions gratuitement d'une abstraction d'un programme fonctionnel à travers sa représentation en grammaire attribuée. En effet, l'analyse statique des durées de vie des attributs (représentant des variables dans la version fonctionnelle) donne des résultats exacts sur un très large sous-ensemble d'attributs de la grammaire attribuée d'entrée. Il nous reste alors, dans notre contexte, à calculer la propriété de partage pour pouvoir déterminer des conditions suffisantes autoraisant un remplacement. Nous avons défini une condition de *destructibilité* qui utilise la notion de durée de vie des attributs et un nouveau concept de partage de structures pour les grammaires attribuées. Le calcul de cette propriété de partage se fait à l'aide d'une analyse statique sur la grammaire d'entrée. Ce type d'analyse est classique et fréquemment utilisé dans les grammaires attribuées ; elle utilise le même type d'abstraction des règles sémantiques (donc du programme fonctionnel) que l'analyse statique sur les durées de vie des attributs. En effet, l'ensemble des méthodes d'analyse statique introduites dans les grammaires attribuées prennent en entrée uniquement une représentation des dépendances entre attributs (communément appelé graphe de dépendance). Ainsi, l'abstraction d'une règle sémantique ne représente que les dépendances entre l'attribut défini et les attributs arguments de cette fonction. Notre analyse de partage de structure sur les attributs utilise cette abstraction. Il est clair que cette modélisation donne les limites de précision de l'information calculée.

1. Toute grammaire attribuée non-circulaire peut être exprimée par un programme fonctionnel.

Ce travail s'inscrit dans un axe de recherche consistant à exploiter des similitudes entre des méthodes développées en grammaires attribuées et dans les langages applicatifs pour résoudre des problèmes voisins.

Le présent article se décompose comme suit : un rapide état de ces recherches en programmation fonctionnelle, une présentation sommaire des grammaires attribuées à l'aide d'un exemple, la définition de notre concept de destructibilité en utilisant les notions de durée de vie et de partage des attributs, la transformation d'une fonction dans sa version destructive et enfin une conclusion présentant les suites possibles de nos recherches.

2 Travaux existants

Dans les langages fonctionnels, la "réutilisation de mémoire en place" a été fréquemment étudiée, étant donné le gain de place et de temps qu'elle peut occasionner. Ce problème a focalisé l'attention sur les conditions dans lesquelles la valeur d'une variable pouvait être physiquement détruite. Il faut en effet que sa modification par une mise à jour destructive ne change pas la sémantique du programme et que cette valeur soit donc inutilisée ultérieurement (directement ou indirectement).

En dehors de la réutilisation même de la place mémoire occupée par la valeur d'une variable, deux concepts très importants sont apparus : la **durée de vie** et le **partage**.

La durée de vie d'une variable est l'intervalle de temps qui sépare sa première définition de sa dernière utilisation. On voit alors qu'une variable ne peut être modifiée physiquement (par une fonction de mise à jour destructive) que si elle est "morte".

D'autre part, les fonctions permettent à des objets complexes de partager entre eux les mêmes espaces physiques, correspondant à des sous-unités communes. Il faut donc prendre garde à ce que, pour deux variables partageant une sous-unité commune, la modification physique de l'une (par une fonction de mise à jour destructive) peut causer, par effet de bord, le changement de valeur de l'autre.

Ces informations devant être connues à la compilation, c'est donc une analyse statique qu'il faut utiliser. L'avènement de l'interprétation abstraite a donné lieu depuis 15 ans à de nombreux travaux dans le domaine de la récupération mémoire au moment de la compilation dans les langages fonctionnels.

Le partage de structures et les conflits entre variables ont été notamment étudiés à l'aide de modèles sémantiques de **comptage de références**. Une interprétation abstraite de ce modèle sur un programme permet de connaître le nombre de références à un ensemble de structures de données. Si un tel ensemble n'est référencé qu'une seule fois, alors sa mise à jour peut être faite de manière destructive plutôt que par recopie des structures [Hud86] [Hed88].

Une autre méthode de détection des conflits potentiels entre les différentes variables d'un programme est l'utilisation de la notion de **chemin** dans un programme. Ce concept de "trace" de l'évolution possible d'un objet, en fonction de ses définitions et de ses utilisations, a donné lieu à diverses représentations (expressions, graphes, grammaires) ; une interprétation abstraite sur ces représentations permet d'obtenir une approximation de l'information recherchée [Blo89] [Hil88] [Mét89] [Ses89].

Un modèle opérationnel des programmes fonctionnels sur lequel on construit une interprétation abstraite permet également d'estimer le partage et la durée de vie de données allouées dynamiquement [Deu90] [Deu92]. La notion de "paire de chemins d'accès symboliques" permet d'obtenir des résultats précis quant à la structure des objets partagés [Deu94].

Ces méthodes utilisant l'interprétation abstraite fournissent des approximations sur le partage et sur la durée de vie des variables dans un programme fonctionnel et permettent dans certains cas d'effectuer statiquement de la récupération de mémoire.

3 Présentation du problème

Dans ce chapitre nous allons préciser notre objectif. Grâce à l'analyse des durées de vie effectuée par le générateur d'évaluateur, un attribut donné peut être implanté en pile ou en variable globale. Par exemple, dans le cas "variable globale", l'ensemble des valeurs des instances d'un même attribut (pour un arbre donné) sont stockées dans la même variable.

Ainsi, une définition d'attribut de la forme $\$a(X) := f(\$b(Y))$ (une règle sémantique) peut être traduite dans le code produit par l'affectation $v := f(v, \dots)$. Notre problème consiste alors à déterminer les conditions dans lesquelles le remplacement de la fonction f par son équivalent destructif ne modifie pas la sémantique du programme (l'affectation ci-dessus étant remplacée par l'appel de procédure $f_d(v, \ldots)$). Par **fonction destructive**, nous entendons qu'elle peut effectuer des modifications physiques sur v. Par exemple, si f représente une fonction d'insertion dans un arbre binaire de recherche qui, dans sa forme originelle, duplique le chemin de la racine jusqu'au point d'insertion, la version destructive réutilise directement l'arbre d'entrée, en le modifiant physiquement au point d'insertion.

La validité (correction) de ce remplacement, dans notre contexte, est liée au partage éventuel de la valeur v d'entrée de f. Les modifications physiques induites par la version destructive de f ne doivent affecter que la valeur de v. Elles ne doivent pas, par effet de bord, changer les valeurs des variables éventuellement partagées avec v.

Pour cela, nous allons introduire un concept de **destructibilité**, fondé sur notre notion de durée de vie et qui est calculé sur le même type d'abstraction des règles sémantiques. Pour décrire précisément ce mécanisme, nous devrions donner les définitions formelles des grammaires attribuées (le type d'abstraction utilisé) et de l'ensemble des méthodes (algorithmes du générateur d'évaluateurs) nécessaires pour ce calcul de durée de vie des attributs. Ce n'est pas notre but ici (se reporter à [Dur94] [Jul89] [JP90] pour les détails).

De façon intuitive, les données d'entrée du générateur d'évaluateurs ne sont qu'une abstraction du programme. Elles se composent de la partie structurelle (la syntaxe du terme accepté en entrée) et, pour chaque constructeur, d'une partie sémantique définie par le graphe de dépendances entre les attributs. L'abstraction de la définition d'un attribut se limite aux dépendances entre celui-ci et les arguments de la règle sémantique.

Aussi nous allons simplement, à l'aide d'un exemple de grammaire attribuée [Joh87] (cf. la figure 1) et de son équivalent en programmation fonctionnelle (cf. figure 2), donner l'idée des divers calculs effectués par le générateur d'évaluateurs en représentant le résultat produit (l'évaluateur) par un programme fonctionnel équivalent (cf. figure 3). Cet exemple traite de la construction, à partir d'un arbre binaire dont les feuilles sont étiquetées par des entiers, d'un arbre ayant la même structure mais dont toutes les feuilles ont la valeur minimale parmi celles des feuilles de l'arbre d'entrée.

Avant de poursuivre, faisons deux remarques. La présentation du générateur d'évaluateurs comme un transformateur de programmes fonctionnels n'est qu'une analogie informelle. Comme ce n'est pas le sujet traité ici, nous ne préciserons pas l'ensemble des conditions dans lesquelles cette analogie est possible.

D'autre part, l'aspect déclaratif des grammaires attribuées trouve tout son intérêt pour une spécification de grande taille (grand nombre d'attributs et structure syntaxique importante). Le générateur d'évaluateurs effectue dans ce cas une transformation de programme non triviale (qui est bien sûr évidente sur notre petit exemple).

Le calcul de durée de vie des attributs qui est effectué par le générateur d'évaluateur peut être interprété en termes fonctionnels comme une évaluation des durées de vie des variables du second programme (figure 3). Par exemple, l'optimiseur mémoire du générateur d'évaluateur détermine que l'attribut *$rmin* peut être stocké en variable globale. Dans la représentation fonctionnelle, il correspond à la variable m de la fonction *replace*.

Nous allons maintenant décrire comment calculer la propriété de destructibilité associée à un attribut donné. Cette analyse statique va utiliser la même abstraction que celle du générateur d'évaluateurs. De plus, comme la destructibilité est calculée à partir des durées de vie des attributs, cette analyse s'effectuera dans le générateur d'évaluateurs, en aval de l'analyse des durées de vie [Par88] [JPJ+90].

4 Test de destructibilité

Etant donné l'abstraction que nous utilisons, lorsqu'un attribut est défini à partir (dépend) d'un autre attribut, nous dirons qu'il y a *partage* entre eux. Il est évident que cette abstraction donne une solution approchée, mais elle est sûre.

La propriété de destructibilité est calculée localement sur chaque abstraction de définition d'attribut (dépendance). Nous supposons au départ que tous les attributs sont destructibles.

Prenons une définition d'attribut de la forme $a_0 := f(a_1, \ldots, a_n)$.

Il est clair que la non-destructibilité de l'un des a_i entraîne la non-destructibilité de a_0. En effet, en l'absence d'autre information sur f, il faut supposer que la valeur de a_i, ou l'une de ses sous-unités, se retrouvera dans la valeur de a_0 ; s'il est interdit de modifier la valeur de a_i, il doit en être de même pour a_0.

Par ailleurs, si l'un des a_i est encore en vie après son utilisation par f, alors l'attribut a_0 devient non destructible ; en effet, l'utilisation ultérieure de cet a_i peut provoquer un partage.

En résumé, a_0 devient non destructible dès que l'un des a_i n'est pas dans sa dernière utilisation (est encore en vie) ou qu'il n'est pas destructible.

```
{ L'attribut $min calcule la valeur minimale des feuilles du bas vers le haut.}
{ Cette valeur est propagée dans l'arbre par l'attribut $rmin.               }
{ A l'aide de cette valeur, l'attribut $tree reconstruit l'arbre.            }

   attribute grammar minimal-tree(BIN-TREE) : BIN-TREE is
   attribute
        synthesized $min(TREE) : int ;
        synthesized $tree(TREE) : TREE ;
        synthesized $axiom-tree(BIN-TREE) : BIN-TREE ;
        inherited $rmin(TREE) : int ;

   where bin-tree -> TREE use          { Règle ``axiome'' nécessaire en G.A. }
        $rmin(TREE) := $min(TREE) ;
        $axiom-tree(bin-tree) := bin-tree($tree(TREE)) ;
   end where ;

   where fork -> LEFT RIGHT use        { Production d'un noeud de l'arbre. }
        $min(fork) := min ($min(LEFT) , $min(RIGHT)) ;
        $rmin(LEFT) := $rmin(fork) ;
        $rmin(RIGHT) := $rmin(fork) ;
        $tree(fork) := fork($tree(LEFT),$tree(RIGHT)) ;
   end where ;

   where tip -> VAL use                { Production d'une feuille de l'arbre. }
        $min(tip) := VAL ;
        $tree(tip) := tip($rmin(tip)) ;
   end where ;

   end grammar ;
```

FIG. 1 - *Programme* OLGA *construisant l'arbre minimum.*

```
transform t = t1 where rec (t1, m1) = repmin t m1
where rec
     repmin (tip n) m = (tip m, n) ||
     repmin (fork L R) m = (fork t1 t2, min m1 m2)
            where (t1, m1) = repmin L m
              and (t2, m2) = repmin R m
```

FIG. 2 - *Programme ML (paresseux) correspondant à la grammaire attribuée.*

```
transform t = replace t (tmin t)
where rec
     replace (tip n) m = tip m ||
     replace (fork L R) m = fork (replace L m) (replace R m)
and
     tmin (tip n) = n ||
     tmin (fork L R) = min (tmin L) (tmin R)
```

FIG. 3 - *Programme ML correspondant à l'évaluateur engendré.*

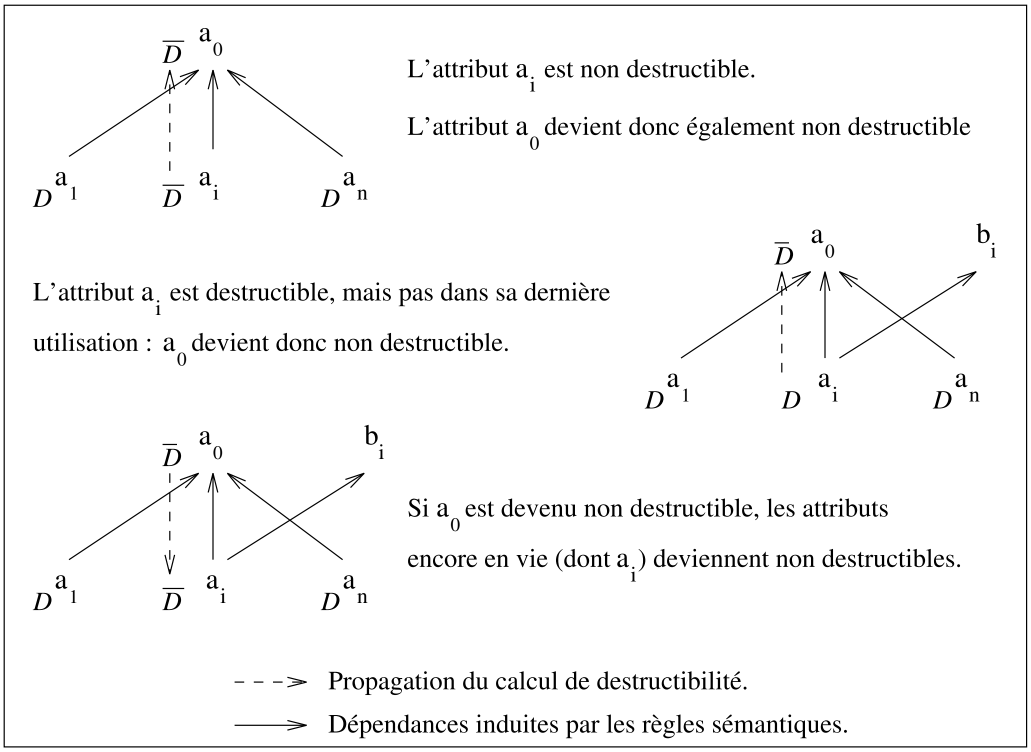

FIG. 4 - *Propagation de l'information de destructibilité.*

D'autre part, lorsqu'a_0 devient non destructible, tous les a_i qui sont encore en vie deviennent non destructibles. En effet, ces a_i seront utilisés ultérieurement pour définir d'autres attributs (notons-les b_i). a_0 et b_i risquent alors de partager a_i. Les a_i encore en vie permettent de véhiculer l'information de partage et de rendre à terme les b_i non destructibles.

Par contre, lorsqu'a_0 devient non destructible, il n'est pas nécessaire de rendre non destructibles les a_i qui sont dans leur dernière utilisation, puisqu'ils n'auront plus l'occasion d'être partagés.

La figure 4 représente des schémas associés à ces différentes propagations de l'information de destructibilité. La seule façon pour un attribut d'être destructible est de ne dépendre que d'attributs qui soient destructibles et dans leur dernière utilisation.

Cette notion statique de destructibilité est calculée localement, au niveau des dépendances (règles sémantiques), et jusqu'à l'obtention de la stabilité de l'information (point fixe) sur l'ensemble des attributs de la grammaire attribuée. Ce processus de calcul d'une propriété sur des attributs est classique et couramment utilisé en grammaires attribuées [Jou92].

Comme nous l'avons dit plus haut, notre notion de partage (associée à la dépendance induite par une règle sémantique) est approchée et certainement trop restrictive. Cependant, il est possible d'affiner cette notion en utilisant des propriétés de type. Le partage entre deux attributs ne peut avoir lieu que si certaines adéquations de types sont vérifiées (a_0 ne peut partager a_i que si a_i est d'un type compatible avec une (sous-)structure de a_0). Une étude de type permet donc de restreindre les profils des règles sémantiques (les dépendances) pour le calcul de la destructibilité des attributs et d'affiner l'information calculée (notamment avec la notion de "types influencés" par une fonction [Dur94]).

5 Transformation

Nous savons maintenant dans quelles conditions le remplacement d'un appel à une fonction applicative par un appel à sa version destructive est correct. Reste maintenant à construire automatiquement la version destructive d'une fonction.

Les fonctions qu'il nous intéresse de transformer sont celles qui effectuent des mises à jour sur des objets structurés. Elles sont en fait utilisées dans les règles sémantiques définissant les attributs. Elles sont normalement écrites dans un langage fonctionnel, mais nous pouvons supposer qu'en pratique, dans les applications que nous traitons, elles vérifient les conditions suffisantes permettant de les exprimer sous la forme de grammaires attribuées dont le résultat est l'attribut défini par la règle sémantique utilisant cette fonction.

Dans le contexte des grammaires attribuées, et plus particulièrement du système FNC-2 [JPJ+90], nous avons donc simplifié le problème de la construction automatique de la version destructive d'une fonction de mise à jour en considérant que nous disposions de la fonction à transformer sous la forme d'une grammaire attribuée, écrite dans notre langage de spécification OLGA.

Cette forme nous offre une spécification précise de la structure de l'argument d'entrée et du résultat construit. D'autre part, la structure guidée par la syntaxe (par filtrage) d'un programme OLGA permet d'accéder aux productions (constructeurs) qui construisent le résultat de la grammaire attribuée.

L'avantage principal de cette restriction (FNC-2, OLGA) est de disposer d'un formalisme syntaxique permettant d'analyser précisément un objet (l'arbre d'entrée) et de pouvoir, au fil de la construction du résultat, en réutiliser des parties. D'autre part, notre système FNC-2 peut lui-même générer le code de la version destructive, grâce à un algorithme qui provoque, lors de la génération de l'évaluateur associé à la grammaire attribuée, des affectations physiques sur les nœuds de l'arbre d'entrée. Ces affectations ne peuvent avoir lieu que sous certaines conditions dépendant à la fois de la structure construite (mapping syntaxique) et de l'ordre d'évaluation des attributs (qui est alors connu par le générateur d'évaluateurs). Cet algorithme de transformation permet de réutiliser des parties de la structure de l'arbre d'entrée et ainsi d'éviter des allocations et des désallocations d'espace mémoire. Le détail de cette transformation est explicité dans [Dur94].

De cette façon, c'est le système FNC-2 qui produit l'évaluateur correspondant à la version destructive d'une fonction de mise à jour applicative et le code correspondant peut être mis en place ou non dans l'application générale en fonction de la destructibilité de l'argument d'entrée (calculée par notre algorithme de décision).

6 Conclusion

Nous avons présenté une transformation de grammaire attribuée qui peut être vue, sous certaines conditions, comme une transformation du programme fonctionnel équivalent. L'objectif de départ aurait pu être atteint par les techniques déjà développées pour les langages fonctionnels. Nous avons montré que les méthodes classiques pour les grammaires attribuées permettent d'obtenir un résultat comparable.

De notre point de vue, l'utilisation de nos techniques pour réaliser ce type de transformation présente les intérêts suivants. D'une part, nous montrons que les techniques d'analyse statique, largement répandues dans le domaine des grammaires attribuée, peuvent être vues comme une autre méthode de mise à jour destructive que celles déjà connues en programmation fonctionnelle. D'autre part, nous avons étendu notre analyse sur les durées de vie avec la notion de destructibilité pour obtenir une méthode complète d'*update in place*. Cette extension était relativement facile à mettre en œuvre, étant donné les abstractions et les informations déjà calculées par notre générateur d'évaluateurs.

Enfin, ces deux derniers point laissent à penser qu'il existe dans la théorie des grammaires attribuées d'autres résultats que l'on peut interpréter en termes fonctionnels. Par exemple, sous certaines conditions [Dur94], la "composition descriptionnelle" des grammaires attribuées [Rou94] est comparable à la "déforestation" dans les programmes fonctionnels [Wad88].

Nous ne prétendons pas que ces résultats sont utilisables de façon immédiate, et ce pour les raisons suivantes. D'une part, nos méthodes ont été introduites pour répondre à des objectifs qui ne recouvrent pas exactement ceux des domaines applicatifs. Une éventuelle utilisation de nos méthodes demandera certainement une extension comme celle que nous avons introduite pour étendre la durée de vie des attributs en une notion de destructibilité. D'autre part, l'interprétation d'une grammaire attribuée comme une abstraction d'un programme fonctionnel est sujette à des restrictions, étant donnée la nature initiale des grammaires attribuées.

Cependant, nous sommes persuadés que nous pouvons abstraire en partie cette vision initiale des grammaires attribuées (prédominance de la structure arborescente) de façon à utiliser nos divers résultats dans un cadre plus large, sans remettre fondamentalement en cause les méthodes existant actuellement dans notre générateur d'évaluateur.

Références

[Blo89] Adrienne Bloss. Update analysis and the efficient implementation of functional aggregates. In *Conf. on Func. Prog. Languages and Computer Architecture.*, pages 26–38, London, September 1989.

[Deu90] Alain Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *ACM SIGPLAN '84 Symp. on Compiler Construction*, pages 157–168, January 1990.

[Deu92] Alain Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *Proc. of the IEEE International Conference on Computer Languages*, San Francisco, April 1992.

[Deu94] Alain Deutsch. Interprocedural may-alias analysis for pointers : Beyond k-limiting. In *ACM SIGPLAN '94 Symp. on Programming Languages Design and Implementation*, June 1994.

[DJL88] Pierre Deransart, Martin Jourdan, and Bernard Lorho. *Attribute Grammars: Definitions, Systems and Bibliography*, volume 323 of *Lect. Notes in Comp. Sci.* Springer-Verlag, August 1988.

[Dur94] Étienne Duris. Transformation de grammaires attribuées pour des mises à jour destructives. Rapport de DEA, Université d'Orléans, September 1994.

[Hed88] Lucy Hederman. Compile time garbage collection using reference count analysis. Technical report, Rice University. Houston, Texas, 1988.

[Hil88] James R. Larus & Paul N. Hilfinger. Detecting conflicts between structure accesses. In *SIGPLAN Conf. on Programming Languages Design and Implementation*, Atlanta, Georgia, June 1988.

[Hud86] Paul Hudak. A semantic model of reference counting and its abstraction. In *ACM Symp. on LISP and Functional Programming*, pages 351–363, August 1986.

[Joh87] Thomas Johnsson. Attribute grammars as a functional programming paradigm. In Gilles Kahn, editor, *Conf. on Functional Prog. Languages and Computer Architecture*, volume 274 of *Lect. Notes in Comp. Sci.*, pages 154–173, Portland, OR, September 1987.

[Jou92] Martin Jourdan. *Des bienfaits de l'analyse statique sur la mise en œuvre des grammaires attribuées.* Mémoire d'habilitation, Département de Mathématiques et d'Informatique, Université d'Orléans, April 1992.

[JP90] Catherine Julié and Didier Parigot. Space Optimization in the FNC-2 Attribute Grammar System. In Pierre Deransart and Martin Jourdan, editors, *Attribute Grammars and their Applications (WAGA)*, volume 461 of *Lect. Notes in Comp. Sci.*, pages 29–45, Paris, September 1990. Springer-Verlag.

[JPJ+90] Martin Jourdan, Didier Parigot, Catherine Julié, Olivier Durin, and Carole Le Bellec. Design, Implementation and Evaluation of the FNC-2 Attribute Grammar System. In *ACM SIGPLAN '90 Conf. on Programming Languages Design and Implementation*, pages 209–222, White Plains, NY, June 1990.

[Jul89] Catherine Julié. *Optimisation De L'Espace MÉMoire Pour L'ÉValuation De Grammaires AttribuÉEs.* PhD thesis, Université d'Orléans, September 1989.

[Mét89] Simon B. Jones & Daniel Le Métayer. Compile-time garbage collection by sharing analysis. In *Conf. on Func. Prog. Languages and Computer Architecture.*, pages 54–74, London, September 1989.

[Par88] Didier Parigot. *Transformations, évaluation incrémentale et optimisations des grammaires attribuées: le système FNC-2.* PhD thesis, Université de Paris-Sud, Orsay, May 1988.

[Rou94] Gilles Roussel. *Algorithmes de base pour la modularité et la réutilisabilité des grammaires attribuées.* PhD thesis, Département d'Informatique, Université de Paris 6, March 1994.

[Ses89] Peter Sestoft. Replacing function parameters by global variables. In *Conf. on Func. Prog. Languages and Computer Architecture.*, pages 39–53, London, September 1989.

[Wad88] Philip Wadler. Deforestation: Transforming Programs to Eliminate Trees. In Harald Ganzinger, editor, *European Symposium on Programming (ESOP '88)*, volume 300 of *Lect. Notes in Comp. Sci.*, pages 344–358, Nancy, March 1988.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399