Les grammaires attribuées : un langage fonctionnel déclaratif

Didier PARIGOT[1], Gilles ROUSSEL[2], Etienne DURIS[1] & Martin JOURDAN[1]

1: INRIA, Domaine de Voluceau, Rocquencourt,
BP 105, 78153 Le Chesnay Cedex, France,
{Didier.Parigot,Etienne.Duris,Martin.Jourdan}@inria.fr
2: Université de Marne-la-Vallée
2, allée du Promontoire, 93166 Noisy-le-Grand, France,
roussel@univ-mlv.fr

Résumé

Bien que les grammaires attribuées aient été introduites il y a trente ans, leur manque de pouvoir d'expression les a confinées dans le domaine du traitement des langages de programmation. Dans cet article, nous montrons qu'il est possible d'étendre cette expressivité. Nous soutenons que les grammaires attribuées peuvent être utilisées pour décrire des calculs sur des structures qui ne sont pas uniquement des arbres, mais aussi des formes abstraites permettant de décrire des structures infinies. Afin d'atteindre cette expressivité, nous avons introduit deux nouvelles notions : les *schémas de productions* et les *productions conditionnelles*. Nous obtenons ainsi un langage dont le pouvoir d'expression est comparable à celui de la plupart des langages fonctionnels du premier ordre, avec un côté déclaratif beaucoup plus marqué.

Nos extensions ne remettent pas en cause les bases du formalisme des grammaires attribuées sur lesquelles reposent la plupart des travaux concernant celles-ci, en particulier l'analyse statique et la génération d'évaluateurs. Ainsi, les résultats existants peuvent être appliqués directement à nos grammaires attribuées étendues, entre autre ceux permettant une implantation efficace (dans notre cas, en utilisant notre système FNC-2).

L'intérêt de ces extensions est de redonner aux grammaires attribuées leur expressivité intrinsèque. De plus, elles nous permettent d'envisager de nouveaux axes de recherche en comparant nos techniques d'analyses à celles qui ont été développées dans des formalismes de même expressivité.

1. Introduction

Il y a trente ans, Knuth [20] introduisait les grammaires attribuées. Depuis, elles ont été largement étudiées [7, 6, 1]. Une grammaire attribuée est une spécification déclarative qui décrit comment des attributs (variables) sont calculés sur chaque règle d'une syntaxe particulière. À l'origine, les grammaires attribuées ont été introduites comme un formalisme permettant de spécifier des applications liées à la compilation ; elles étaient destinées à décrire comment décorer un arbre et ne pouvaient être utilisées sans une structure sous-jacente (arbre) représentant un programme à traiter. Dans ce type d'applications, deux qualités importantes sont reconnues aux grammaires attribuées :

- elles permettent très naturellement une *décomposition structurelle* qui correspond à la structure syntaxique du langage, et
- elle sont *déclaratives* dans le sens où l'écrivain spécifie uniquement les règles à utiliser pour calculer la valeur des attributs, et jamais dans quel ordre doivent être effectués ces calculs.

La question importante sur ce formalisme fut alors : "Est-il possible de produire un code exécutable à partir d'une spécification par grammaires attribuées ?" La plupart des recherches dans ce domaine se sont donc focalisées sur la production automatique d'un code efficace et de très bons résultats ont été obtenus, en particulier des techniques d'implantation efficaces et des méthodes d'analyse statique globale qui sont généralement très précises [14]. En dépit de cela, les grammaires attribuées ne sont toujours pas utilisées aussi largement qu'elles le pourraient. Elles sont invariablement considérées comme une partie inhérente d'un processus de compilation.

Une des raisons de la faible utilisation des grammaires attribuées est leur manque d'expressivité. Il n'y a que peu de domaines qui disposent naturellement en entrée d'un arbre qui dirige les calculs. Sans arbre, le modèle de calcul des grammaires attribuées est inutilisable [1]. Ce manque d'expressivité, ainsi que des désaccords sur des concepts de base au sein de la communauté des grammaires attribuées, ont contribué à masquer les bons résultats accumulés depuis trente ans. Cela a conduit certains à penser que les grammaires attribuées étaient si simples et limitées que les méthodes utilisées pour leur analyse statique ne pouvaient pas être transposées à d'autres formalismes.

Des travaux ont essayé de répondre à ce problème en proposant diverses extensions au formalisme original des grammaires attribuées,

1. Il n'est même pas possible de spécifier la fonction factorielle avec une grammaire attribuée classique

par exemple les grammaires attribuées d'Ordre Supérieur [29], les grammaires attribuées Circulaires [10] ou les Grammaires Multi-Attribuées [2]. Nous avons le même but, mais la différence la plus marquante entre ces travaux et les nôtres est la méthodologie utilisée pour aborder le problème. En effet, pour chacune d'elles, la première étape a été de proposer une extension syntaxique permettant d'exprimer plus simplement une application particulière (par exemple, l'analyse de flot de données pour les grammaires attribuées Circulaires). Ce n'est qu'ensuite que sont résolues les questions concernant leur implantation. Au contraire, notre approche a été, premièrement, de caractériser précisément le pouvoir d'expression du formalisme original et, ensuite seulement, de définir les extensions au langage qui permettent de l'exploiter entièrement.

Notre remarque fondamentale est que la notion de *grammaire* n'implique pas nécessairement l'existence (physique) d'un *arbre*. Plus précisément, notre interprétation de la grammaire sous-jacente à une grammaire attribuée est la même que celle de la grammaire décrivant tous les arbres d'appels pour un programme fonctionnel donné ou de tous les arbres de preuve pour un programme logique donné : la grammaire décrit l'ensemble des flots de contrôle possibles. Dans ce contexte, une production décrit un schéma récursif élémentaire [5] (flot de contrôle), tandis que les règles sémantiques décrivent les calculs associés à ce schéma (flot de données).

Il est très important à ce point de la discussion de remarquer que la plus grande partie des résultats théoriques et pratiques concernant les grammaires attribuées, en particulier les algorithmes de construction d'évaluateurs efficaces, sont fondés *uniquement* sur une abstraction du flot de contrôle grâce à la grammaire, et pas du tout sur la manière dont une instance particulière de ce flot est obtenue au cours d'une exécution.

En conséquence, nous proposons deux extensions syntaxiques qui sont en accord avec cette vision :

- les *productions conditionnelles* qui permettent de diriger le flot de contrôle grâce à des valeurs d'attributs, et
- les *schémas de production* qui permettent de décrire un schéma de récursion indépendamment de toute structure physique et/ou permettent une combinaison différente des éléments d'une structure physique (production) donnée.

Ces extensions permettent de réfuter la critique majeure faite aux grammaires attribuées, c'est-à-dire leur manque d'expressivité. Elles définissent un langage de programmation comparable à un langage du premier ordre, avec un aspect fonctionnel (à cause de l'affectation

unique), tout en gardant le caractère de déclarativité propre aux grammaires attribuées.

Proposer ce "nouveau" langage n'est pas le but principal de notre travail. En effet, nous pensons que la comparaison entre diverses formes externes de langages est un exercice très subjectif. L'intérêt de ce langage est qu'il représente, pour nous, la forme externe intrinsèque permettant d'exploiter les techniques d'analyse statique développées pour les grammaires attribuées (qui sont dignes d'intérêt par elles-même). En libérant les grammaires attribuées de leur cadre classique, nous pensons qu'elles trouveront plus facilement des applications dans d'autres domaines.

Cet article est divisé en trois sections. Dans la première, les notions de production conditionnelle et de schéma de production sont introduites au moyen d'exemples simples, choisis intentionnellement en dehors du domaine de la compilation. La section suivante montre pourquoi ces extensions n'entraînent pas de modifications des techniques utilisées dans le générateur d'évaluateur. La dernière section énumère les intérêts liés à l'utilisation de cette nouvelle forme externe des grammaires attribuées. Cette présentation est volontairement informelle.

2. Vue externe des grammaires attribuées

Dans cette section nous allons décrire les grammaires attribuées classiques et nos extensions à l'aide d'exemples simples, spécifiés dans le langage de description de grammaires attribuées Olga [16]. Afin de donner une idée intuitive de la sémantique des grammaires attribuées classiques et de nos extensions, certains exemples seront aussi donnés dans le langage fonctionnel CAML. La référence [25] propose des exemples plus nombreux.

2.1. La vue classique des grammaires attribuées

Nous introduisons le formalisme classique des grammaires attribuées à l'aide d'un exemple simple, *bird* [4, 12]. Il prend en entrée un arbre binaire avec des entiers aux feuilles, et donne en sortie un arbre binaire de même structure dont toutes les feuilles ont été remplacées par la valeur minimale des feuilles de l'arbre d'entrée.

Avant de donner la spécification par grammaire attribuée, nous spécifions la structure de l'arbre, sa syntaxe abstraite. Cette syntaxe

abstraite peut aussi être décrite sous forme d'un type ML. [2]

Syntaxe Abstraite :

```
AXIOM = axiom;
axiom -> TREE;
TREE = fork tip;
fork -> TREE TREE;
tip -> int;
```

Type ML :

```
type TREE = fork of TREE * TREE | tip of int;;
```

L'attribut `$min` est utilisé pour calculer la valeur minimale des feuilles de l'arbre d'entrée, l'attribut `$n` propage cette valeur minimale à travers l'arbre et l'attribut `$tree` construit l'arbre de sortie.

Spécification par grammaire attribuée :

```
attribute grammar bird (TREE) : (TREE) is
attribute
    synthesized $min (TREE) : int;
    synthesized $tree (TREE) : TREE;
    synthesized $tree_axiom (AXIOM) : AXIOM;
    inherited $n (TREE) : int;

where axiom -> T use
    $tree_axiom := axiom($tree(T)); $n (T) := $min (T);
end where;

where tip -> M use
    $min := M; $tree := tip ($n(tip)) ;
end where;

where fork -> L R use
    $min := if ($min(L) <= $min(R))
            then $min(L) else $min(R) end if;
    $tree := fork ($tree(L), $tree(R));
    $n(L) := $n(fork); $n(R) := $n(fork);
end where;
end grammar;
```

Une transformation syntaxique simple de cette grammaire attribuée [12]

2. La spécification par Grammaire Attribuée exige la notion de racine qui n'est pas nécessaire dans le cas de ML.

résulte en l'unique fonction ML suivante :

ML:
```
let bird t = t1
    where rec (t1, n) = repmin t n
    where rec repmin t n = match t with
       tip (m) -> (tip (n), m) |
       fork (L, R) -> let (t1, m1) = repmin L n
                      and (t2, m2) = repmin R n
                      in (fork (t1, t2),
                          (if m1 <m2 then m1 else m2));;
```

On peut noter que la forme ML requiert une évaluation paresseuse ; ceci est dû à la "circularité" sur l'argument n (valeur minimale des feuilles) au niveau de la racine. L'argument n est à la fois résultat et argument de la fonction.

Notre système FNC-2 [3] peut produire des évaluateurs dans divers langages dont ML [3]. Dans la figure suivante, nous présentons la forme ML équivalente à l'évaluateur qui est produit automatiquement à partir de la grammaire attribuée *bird*. Cette forme ne nécessite pas d'évaluation paresseuse, mais nous semble moins naturelle : l'écrivain doit écrire deux fonctions de parcours d'arbre pour spécifier la solution.

Le programme ML dérivé de la Grammaire Attribuée :
```
let rec replace t m = match t with
    tip (n) -> tip(m) |
    fork(L, R) -> fork ((replace L m), (replace R m));;

let rec tmin t = match t with
    tip (n) -> n |
    fork (L, R) -> let m1 = (tmin L) and m2 = (tmin R)
                   in (if m1 < m2 then m1 else m2);;

let bird t = replace t (tmin t);;
```

Dans la Grammaire Attribuée, l'ordre d'évaluation n'est jamais spécifié, ce qui est l'essence même de la notion de déclarativité. En effet, le générateur d'évaluateur détermine automatiquement qu'il faut d'abord calculer la valeur minimale, puis construire l'arbre. Ce découpage en deux passes élimine la circularité. Sur cet exemple simple, l'ordre d'évaluation est très facile à trouver "à la main", mais sur de gros exemples ce peut être beaucoup moins évident.

3. Voir `http://www-rocq.inria.fr/charme/FNC-2/`

2.2. Les productions conditionnelles

Dans cette section, nous allons introduire la notion de *production conditionnelle*. La syntaxe proposée ici, qui introduit une sorte de "garde" sur les productions, est une forme très simple. Par la suite, nous en présenterons une forme plus complète. Ces nouvelles productions permettent de spécifier différents ensembles d'équations (règles sémantiques) pour une unique production syntaxique. Le choix de l'application de telle ou telle production conditionnelle est fait dynamiquement en fonction de la valeur de la condition (garde). Essentiellement, nous avons ajouté des prédicats, calculables à partir de valeurs d'attributs, à la notion classique de sélection en fonction des productions pour diriger les calculs.

Dans l'exemple suivant, il existe deux ensembles de règles sémantiques différentes sur la production `fork`. La valeur de l'attribut `$min` permet de choisir entre les deux ensembles d'équations. Cette spécification a la même sémantique que la précédente. La présence d'une production conditionnelle par défaut, indiquée par la condition vide `[]`, est obligatoire. Elle sera appliquée dans le cas où aucune autre production conditionnelle ne peut être appliquée.

Production Conditionnelle :

```
where fork[$min(L) <= $min(R)] -> L R use
    $min := $min(L);
end where;

where fork[] -> L R use
    $min := $min(R);
end where;
```

Nous verrons dans la suite de cet article d'autres exemples de la notion de production conditionnelle. L'intérêt de celles-ci deviendra clair lors de leur combinaison avec les schémas de production. Notons cependant qu'utilisées seules, les productions conditionnelles ont l'intérêt de découpler les différents systèmes de dépendances des diverses alternatives, permettant une écriture et une analyse plus simples.

2.3. Les Schémas de Production

Dans la théorie classique des grammaires attribuées, la notion de grammaire sous-entend l'existence d'un arbre. Nous allons étendre cette notion de grammaire à l'aide de la notion de schéma de production. Les schémas de production permettent de décrire comment vont se connecter dynamiquement plusieurs ensembles de règles sémantiques ;

dans le formalisme classique des grammaires attribuées, cette connexion est exclusivement dirigée par la structure de l'arbre d'entrée.

Avec la notion de production conditionnelle, les schémas de production permettent d'exprimer un grand nombre de problèmes (schémas récursifs primitifs) qui, pour la plupart, n'étaient pas exprimables dans le formalisme classique des grammaires attribuées. Pour présenter l'utilisation de cette extension, nous allons spécifier, en utilisant une grammaire attribuée, la fonction factorielle.

Nous introduisons donc la définition d'un non-terminal qui dérive dans des schémas de productions (virtuels), au lieu de dériver dans des productions classiques (physiques). Ce non-terminal peut être vu comme un type. Le type **FACT** est défini comme pouvant dériver dans deux schémas de production, décrivant les deux appels récursifs possibles. Ces schémas de productions ne définissent pas une structure d'arbre.

Le schéma récursif de la fonction factorielle :

```
FACT = with scheme
  fac-r -> FACT;
  fac-t -> ;
end scheme;
```

Le premier schéma de production définit le cas de l'appel récursif à **FACT** et le deuxième la terminaison de la récursion. Ces schémas définissent entièrement le type **FACT**. Nous verrons plus tard que ces schémas de production sont interprétés dans la grammaire attribuée comme des productions classiques. Les divers ensembles d'équations (règles sémantiques) sont structurés suivant le schéma récursif ou terminal. Une condition sur le nombre **$n** permet de choisir entre ces deux cas.

La fonction factorielle :

```
attribute grammar FACT($n:int) : ($r:int) is
where fac-t [$n <= 1] -> use
  $r := 1;
end where;

where fac-r [] -> F use
  $n(F) := $n - 1;
  $r := $n * $r(F);
end where;
```

Il faut noter que le type **FACT** est un type virtuel, c'est-à-dire qu'il n'existe pas de valeur de ce type et, par conséquent, qu'il n'est pas possible de s'en servir afin de diriger les calculs des attributs. La condition est la seule valeur utilisée pour diriger l'évaluateur.

2.4. Combinaison des extensions avec des productions classiques

Avec les grammaires attribuées classiques, il est impossible de décrire la sémantique dynamique d'un langage contenant des boucles (comme l'instruction **while**). Dans l'exemple suivant, nous étendons la production classique `while` par le schéma de production `wr` qui permet d'introduire une circularité dans l'arbre d'entrée (le troisième élément en partie droite de la production fait référence à la partie gauche).

Syntaxe Abstraite :
```
Expr = while;
while -> C:Cond E:Expr
with scheme
    wr -> C E wr
    wt -> C
end scheme;
```

Dans le second schéma de production `wt`, cas où la condition est devenue fausse, le corps de la boucle est omis, ce qui permet de ne pas spécifier la définition de ses attributs.

Nous décrivons maintenant une partie d'un interprète sur l'instruction `while` sous la forme d'une grammaire attribuée. Les attributs `$s.env` et `$h.env` contiennent l'environnement d'exécution et l'attribut `$c` définit la valeur de la condition.

Spécification de la grammaire attribuée :
```
where wt[$c (C) = false] -> C use
    $h.env(C) := $h.env(wt); $s.env(wt):= $h.env(wt);
end where
where wr[] -> C E W use
    $h.env(C) := $h.env(wr); $h.env(E):= $h.env(wr);
    $h.env(W):= $s.env(E);  $s.env(wr):= $s.env(W);
end where;
```

3. Vision interne des grammaires attribuées

Comme nous l'avons signalé dans l'introduction, nous pouvons aisément incorporer nos extensions du formalisme des grammaires attribuées dans un système déjà existant fondé sur les méthodes d'évaluation à ordre statique, comme FNC-2, sans modification du générateur d'évaluateur. C'est un résultat très intéressant, car le générateur d'évaluateur est la partie la plus compliquée et la plus

importante d'un système de traitement de grammaires attribuées. Notons cependant qu'il est indispensable de disposer d'un ordre total d'évaluation, ce qui nous limite à la classe des grammaires attribuées fortement non circulaires (FNC) [24] ; en pratique, cette restriction n'est pas gênante.

L'idée de base, présentée ici de manière très informelle, est de fournir au générateur d'évaluateur une version de la grammaire attribuée d'entrée dans laquelle les extensions ont été remplacées par leurs effets sur le flot de contrôle. Dans le cas de productions conditionnelles, cela revient à éliminer les conditions et à renommer de façon unique chaque alternative. Pour les schémas de production, aucune transformation n'est nécessaire.

Une fois que le générateur d'évaluateur a produit l'évaluateur correspondant à cette grammaire attribuée sans extension, il est nécessaire de réintroduire la sémantique des extensions dans ce dernier.

Pour un schéma de production, ceci consiste simplement à remplacer les références aux nœuds en partie droite du schéma par des références aux nœuds réels qui leur correspondent (s'il y en a). Dans le cas de schémas circulaires, comme dans l'exemple `while`, cela signifie qu'un nœud donné peut être visité plusieurs fois, éventuellement indéfiniment (ce qui est exactement l'objet de cette extension). Cela oblige l'évaluateur à conserver toutes les valeurs d'attributs en dehors de l'arbre. Le système FNC-2 utilise des techniques d'analyse sophistiquées, fondées sur la durée de vie des attributs [18], qui lui permettent de stocker la plupart des attributs dans des variables globales ou des piles ; en ayant recours à une technique plus systématique comme les *binding trees* [29], les quelques attributs restants peuvent aussi être stockés hors de l'arbre.

Les productions conditionnelles permettent de diriger d'une nouvelle façon le choix entre différents systèmes de définitions d'attributs applicables à un point donné. En tenant compte de cela, le générateur de code du système devra produire un nouveau sélecteur de production. Une production qui a été renommée à partir d'une production conditionnelle est sélectionnée quand, d'une part, la production sous-jacente a été reconnue et que, d'autre part, la condition de la forme étendue est satisfaite. Bien sûr, cela n'est possible que si la valeur de la condition est calculée avant tout autre attribut de la production, et ne doit donc pas en dépendre.

Cette restriction n'est évidemment pas acceptable. Premièrement, cela réduit considérablement le pouvoir d'expression. Deuxièmement, ce n'est pas en accord avec le caractère déclaratif de la spécification parce que cela oblige l'auteur à penser aux dépendances et à l'ordre d'évaluation.

Pour résoudre ces problèmes, nous proposons une autre forme syntaxique pour les productions conditionnelles, appelée *forme factorisée*. Voici ce que pourrait être celle du `while` :

Forme factorisée :
```
where while -> C E use
  $h.env(C):= $h.env(while)
  if [$c(C) = false] then scheme wt -> C use
    $s.env(wt):= $h.env(wt)
  end scheme
  else scheme wr -> C E W use
    $h.env(E):= $h.env(wr); $h.env(W):= $s.env(E);
    $s.env(wr):= $s.env(W);
  end scheme;
end where;
```

Il y a deux caractéristiques importantes à retenir. Premièrement, les conditions ne contrôlent pas toutes les règles sémantiques, ce qui permet aux conditions de dépendre d'attributs calculés dans ces règles. Donc, la condition d'évaluation peut survenir n'importe où dans le processus d'évaluation des attributs et pas uniquement lors de la première visite (appel récursif) d'une production. Deuxièmement, la préséance des diverses conditions est maintenant explicite.

À partir de cette forme factorisée, il est possible de produire une forme "aplatie" proche de celle proposée précédemment, grâce à une transformation distributive. Le générateur de code peut alors récupérer les codes d'évaluation engendrés pour chacune des alternatives et peut les regrouper en une partie commune suivie de différents cas dépendant de la condition ; cela permet d'évaluer la condition ailleurs qu'au début.

La validité de ce procédé a été démontrée par son implantation dans le système FNC-2. Ce travail n'a nécessité que quelques hommes-semaines, le générateur d'évaluateurs n'ayant pas été modifié.

4. Les avantages de cette approche

4.1. Un langage de programmation déclaratif

Les extensions du formalisme classique des grammaires attribuées proposées dans cet article augmentent leur puissance d'expression et débouchent sur la définition d'un langage qui est comparable en puissance à une version d'ordre un de ML. Bien que ce langage n'ait pas pour vocation de remplacer ce dernier, il est intéressant car il garde

toutes les propriétés agréables de la programmation avec les grammaires attribuées :

- les spécifications étendues restent déclaratives, les programmeurs ne se préoccupant jamais de l'ordre d'évaluation ;
- elles sont bien structurées car le paradigme de structuration classique, fondé sur les productions, est étendu aux schémas de production et aux productions conditionnelles.

Ces caractéristiques sont, selon nous, très utiles pour le développement de grosses applications [15]. En particulier, elles permettent à l'écrivain de spécifier une application de manière incrémentale : à partir d'une spécification embryonnaire, l'application complète peut croître et être testée pas à pas sans problème de lisibilité. En outre, la plupart des règles de copies de la forme `$a(X):= $a(Y)` peuvent être engendrées automatiquement et n'apparaissent pas dans le programme ; cela permet au développeur de porter toute son attention sur les calculs importants plutôt que sur de simples transferts d'information.

Ce langage de programmation est immédiatement utilisable avec notre système FNC-2 [17, 16]. FNC-2 intègre des mécanismes puissants pour la composition et la réutilisabilité [22, 21, 28]. C'est un système très souple qui permet de produire des évaluateurs d'attributs exhaustifs ou incrémentaux en C, Lisp, et ML, fonctionnant sur monoprocesseurs ou sur machines multiprocesseurs à mémoire partagée [23] et utilisant des techniques d'optimisation mémoire sophistiquées. Avec les processeurs qui l'accompagnent, FNC-2 peut construire des applications complètes. Mais c'est aussi un système ouvert qui peut s'interfacer avec de nombreux autres outils. Cela permet aux développeurs d'exploiter pleinement la complémentarité des grammaires attribuées avec d'autres paradigmes de programmation (en particulier la programmation fonctionnelle), en les laissant choisir le langage et le système le plus approprié pour chaque partie de leurs applications.

À première vue, permettre à des attributs de transporter des valeurs fonctionnelles (fermetures) et d'appliquer celles-ci dans des règles sémantiques est complètement indépendant du formalisme de base des grammaires attribuées et du travail décrit ici. C'est un premier pas vers la conception d'un langage Olga pleinement fonctionnel, mais cela n'est pas le sujet de notre article.

Le choix d'un langage de programmation plutôt qu'un autre étant très subjectif, ce n'est pas notre objectif ici de convertir les utilisateurs de ML à la programmation par grammaires attribuées. En effet, il est clair pour nous que le formalisme des grammaires attribuées est restreint, et doit uniquement être vu comme une abstraction syntaxique d'une certaine

fonction ; il n'a pas pour vocation de remplacer les autres formalismes.

À notre avis, l'intérêt réel de notre approche n'est pas uniquement le langage mais surtout le modèle de calcul sous-jacent, plus précisément les propriétés qui en permettent une analyse statique très fine. Ce modèle est celui des grammaires attribuées au sens classique, et notre travail a "juste" consisté à étendre son domaine d'application.

4.2. Vers une fertilisation croisée

La "nouvelle" puissance des grammaires attribuées incite à réexaminer l'ensemble des résultats produits par trente années de recherche et de pratique sur celles-ci. Le point de départ de cette relecture est que, lorsque les grammaires attribuées sont considérées comme un dialecte fonctionnel, le processus de génération d'évaluateur peut être interprété comme une transformation particulière de programme fonctionnel ; on trouve l'idée dans les exemples de la section 2, avec la version lazy-ML de *bird* et la version ML produite à partir de la grammaire attribuée.

La simplicité du formalisme des grammaires attribuées, qui auparavant passait pour une faiblesse, nous semble maintenant être sa force. En effet, elle permet d'effectuer un grand nombre d'analyses statiques très précises, voire d'obtenir des résultats exacts comme dans le cas du test de circularité [14]. Par exemple, le générateur d'évaluateur du système FNC-2 effectue une analyse complète de toutes les dépendances entre tous les attributs afin de déterminer la façon la plus efficace de mener l'évaluation. Ce type d'analyse peut être comparé aux techniques d'optimisation dans les langages fonctionnels paresseux. De plus, nos techniques d'optimisation mémoire fondées sur la durée de vie des attributs [18] peuvent être interprétées comme des transformations interprocédurales permettant de réduire l'occupation mémoire d'un programme (passage de paramètres) [19]. Dans les deux cas, ces techniques ont été largement étudiées dans le contexte des grammaires attribuées et sont validées depuis longtemps par leur implantation et leur utilisation dans des systèmes comme FNC-2.

Nous avons récemment commencé à explorer d'autres pistes. Par exemple, des travaux préliminaires [9] semblent montrer que la composition descriptionnelle de grammaires attribuées [11, 27, 28] et la déforestation [30] débouchent sur des résultats équivalents, bien qu'elles opèrent différemment. De plus, l'analyse de durée de vie éclaire d'un jour nouveau les problèmes de mise à jour destructive dans les langages fonctionnels [9, 8]. Il reste du travail pour décider si cette approche est valable.

D'autres paradigmes classiques des grammaires attribuées peuvent certainement trouver une nouvelle interprétation, comme l'évaluation

incrémentale [26, 24],l'évaluation parallèle [13, 23] et nos travaux sur la réutilisabilité et la généricité [22, 21, 28].

Il est clair pour nous que nous ne sommes qu'au début de cette nouvelle lecture et qu'il existe, pour l'instant, plus de questions que de réponses.

5. Conclusion

Dans cet article, nous avons appuyé toute notre argumentation sur le fait que le terme de grammaire n'impliquait pas forcément l'existence d'une structure d'arbre et que le terme attribut ne voulait pas forcément dire décoration de l'arbre. Nous avons présenté deux extensions simples du formalisme des grammaires attribuées qui permettent d'exploiter toute la puissance d'expression contenue dans la théorie des grammaires attribuées. Ces extensions restent en accord avec l'idée sous-jacente aux grammaires attribuées, dans le sens où il est toujours possible de bénéficier des résultats et des techniques qui ont été validées dans ce domaine.

Le but de ces extensions est de montrer que nos outils (générateur d'évaluateurs) sont utilisables et applicables dans un contexte plus large. La programmation par grammaires attribuées (déclarative et structurée) et les techniques existantes de la théorie des grammaires attribuées ont acquis, avec ces extensions, un nouveau degré de généralisation et d'abstraction. Par là même, les grammaires attribuées s'approchent ou sont complémentaires d'autres formalismes comme la programmation fonctionnelle.

Cette approche a un intérêt très pratique puisque les mécanismes nécessaires pour supporter ces extensions font déjà partie du système FNC-2, qui a fait la preuve de son utilisabilité lors du développement d'applications réelles de grande taille. L'implantation de ces extensions dans le système FNC-2 a été très rapide.

Références

[1] Henk Alblas & Bořivoj Melichar, réd. *Attribute Grammars, Applications and Systems*, Prague, juin 1991, *Lect. Notes in Comp. Sci.* 545, Springer-Verlag.

[2] Isabelle Attali. *Compilation de programmes TYPOL par attributs sémantiques.* Thèse de doctorat, Université de Nice, avril 1989.

[3] Gilles Le Bâtard. Réalisation dans le système FNC-2 d'un traducteur vers ML. Rapport de stage de maîtrise, IFI, Université de Marne-la-Vallée, juillet 1995.

[4] R. S. Bird. Using circular programs to eliminate multiple traversal of data. *Acta Informatica*, 21, pages 239–250, 1984.

[5] Bruno Courcelle & Paul Franchi-Zannettacci. Attribute Grammars and Recursive Program Schemes (i & ii). *Theor. Comp. Sci.*, 17 (2 & 3), pages 163–191 et 235–257, 1982.

[6] Pierre Deransart & Martin Jourdan, réd. *Attribute Grammars and their Applications (WAGA)*, Paris, septembre 1990, *Lect. Notes in Comp. Sci.* 461, Springer-Verlag.

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

[8] Étienne Duris, Didier Parigot & Martin Jourdan. Mises à jour destructives dans les grammaires attribuées. Rapport de recherche RR-2686, INRIA, Rocquencourt, octobre 1995.

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

[10] Rodney Farrow. Automatic Generation of Fixed-point-finding Evaluators for Circular, but Well-defined, Attribute Grammars. In *ACM SIGPLAN '86 Symp. on Compiler Construction*, pages 85–98, Palo Alto, CA, juin 1986.

[11] Harald Ganzinger & Robert Giegerich. Attribute coupled grammars. In *ACM SIGPLAN '84 Symp. on Compiler Construction*, pages 157–170, Montréal, juin 1984.

[12] Thomas Johnsson. Attribute grammars as a functional programming paradigm. In *Conf. on Functional Prog. Languages and Computer Architecture*, Portland, OR, septembre 1987, *Lect. Notes in Comp. Sci.* 274, Springer-Verlag, pages 154–173.

[13] Martin Jourdan. A Survey of Parallel Attribute Evaluation Methods. In [1], pages 234–255.

[14] 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, avril 1992.

[15] Martin Jourdan & Didier Parigot. Application Development with the FNC-2 Attribute Grammar System. In *Compiler Compilers '90*, Schwerin, octobre 1990, *Lect. Notes in Comp. Sci.* 477, Springer-Verlag, pages 11–25.

[16] Martin Jourdan, Didier Parigot & Tamás Gaál. *The* FNC-2 *System User's Guide and Reference Manual*, version 1.9. INRIA, Rocquencourt, 1993.

[17] Martin Jourdan, Didier Parigot, Catherine Julié, Olivier Durin & 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, juin 1990.

[18] Catherine Julié & Didier Parigot. Space Optimization in the FNC-2 Attribute Grammar System. In [6], pages 29–45.

[19] Uwe Kastens & M. Schmidt. Lifetime Analysis for Procedure Parameters. In *European Symp. on Programming (ESOP '86)*, Saarbrücken, mars 1986, *Lect. Notes in Comp. Sci.* 213, Springer-Verlag, pages 53–69.

[20] Donald E. Knuth. Semantics of context-free languages. *Math. Systems Theory*, 2 (2), pages 127–145, juin 1968.

[21] Carole Le Bellec. *La généricité et les grammaires attribuées.* PhD thesis, Département de Mathématiques et d'Informatique, Université d'Orléans, novembre 1993.

[22] Carole Le Bellec, Martin Jourdan, Didier Parigot & Gilles Roussel. Specification and Implementation of Grammar Couplings Using Attribute Grammars. In *Programming Language Implementation and Logic Programming (PLILP '93)*, Tallinn, août 1993, *Lect. Notes in Comp. Sci.* 714, Springer-Verlag, pages 123–136.

[23] Bruno Marmol. *La parallélisation et l'optimisation mémoire dans l'évaluation des grammaires attribuées.* Thèse de doctorat, Université d'Orléans, novembre 1994.

[24] Didier Parigot. *Transformations, évaluation incrémentale et optimisations des grammaires attribuées : le système* FNC-2. Thèse de doctorat, Université de Paris-Sud, Orsay, mai 1988.

[25] Didier Parigot, Gilles Roussel, Étienne Duris & Martin Jourdan. *Attribute Grammars: a Declarative Functional Language.* Rapport de recherche RR-2662, INRIA, Rocquencourt, octobre 1995.

[26] Thomas Reps. *Generating Language-Based Environments.* MIT Press, 1984.

[27] Gilles Roussel, Martin Jourdan & Didier Parigot. Coupling Evaluators for Attribute Coupled Grammars. In *5th Int. Conf. on Compiler Construction (CC' 94)*, Edinburgh, avril 1994, *Lect. Notes in Comp. Sci.* 786, Springer-Verlag, pages 52–67.

[28] Gilles Roussel. *Algorithmes de base pour la modularité et la réutilisabilité des grammaires attribuées.* Thèse de doctorat, Département d'Informatique, Université de Paris 6, mars 1994.

[29] S. Doaitse Swierstra & Harald H. Vogt. Higher Order Attribute Grammars. In [1], pages 256–296.

[30] Philip Wadler. Deforestation: Transforming Programs to Eliminate Trees. In *European Symposium on Programming (ESOP '88)*, Nancy, mars 1988, *Lect. Notes in Comp. Sci.* 300, Springer-Verlag, pages 344–358.