Rapport de stage de L3
Un éditeur de grammaires attribuées modulaires
IRISA, Équipe S4, sous la direction d'Éric Badouel

Antoine Amarilli

Résumé

Dans ce stage, on s'intéresse à un nouveau moyen de rendre modulaire les grammaires attribuées. Le travail accompli lors du stage est un prototype permettant la construction graphique de grammaires attribuées modulaires et de documents conformes à ces grammaires, la sauvegarde de ces informations sous forme de code Haskell, et, dans des cas simples, l'évaluation d'attributs en utilisant Haskell.

Table des matières

1 Introduction

1.1 Équipe de recherche

L'IRISA (Institut de recherche en informatique et systèmes aléatoires), située à Rennes sur le campus de Beaulieu, a pour domaines de recherche l'informatique, les mathématiques appliquées, ainsi que le traitement du signal et des images. L'équipe S4 (Synthèse et supervision de systèmes, scénarios) y regroupe des disciplines très diverses : réseaux de Petri, sécurité informatique (secrets concurrents), modélisation statistique des pannes, et grammaires attribuées.

1.2 Sujet

Les grammaires attribuées ont été décrites par Donald E. Knuth [1] afin de fournir un formalisme permettant d'attribuer une sémantique aux grammaires algébriques. Il s'agit d'un concept utilisé en compilation, et qui trouve également des applications dans le cadre de paradigmes nouveaux tels que la programmation intentionnelle.

Plusieurs approches ont été explorées pour rendre de telles structures modulaires : la composition descriptionnelle, dans laquelle on cherche à décomposer le calcul des attributs en l'application successive de plusieurs ensembles de règles sémantiques, et la modularité par aspect, dans laquelle on sépare les attributs en groupes d'attributs indépendants.

Mon sujet de stage s'articule autour du travail de mon maître de stage, Éric Badouel, qui vise à décrire une nouvelle forme de modularité axée cette fois-ci sur les sortes de la grammaire. La grammaire est décomposée en modules comportant chacun un ensemble de sortes et leurs productions associées, et un mécanisme de définitions, d'indirections et de paramètres permettent de passer d'un module à l'autre et de réutiliser de façon paramétrique un module à plusieurs endroits.

1.3 Objectifs

Mon stage avait pour objectif de réaliser un prototype implémentant cette nouvelle forme de modularité. Le prototype permet la création de grammaires modulaires, la production de documents conformes à cette grammaire, et la description de règles sémantiques ; il les sauvegarde sous forme de code Haskell. Il peut être utilisé comme environnement de développement graphique pour construire de tels objets d'une façon plus agréable qu'en écrivant à la main du code répétitif. Au-delà de son utilité intrinsèque, le prototype était également un moyen d'identifier les problèmes subtils que pouvaient poser la formalisation précise de la modularité des grammaires attribuées.

Le prototype est également en mesure de procéder à l'évaluation d'attributs dans des cas simples. L'évaluation d'attributs n'est pas encore possible dans le cadre de grammaires utilisant la modularité, car ce concept n'a pas encore été entièrement formalisé par mon maître de stage.

1.4 Structure du rapport

Dans un premier temps, nous présenterons le formalisme de grammaires attribuées modulaires qui est implémenté par le prototype. Nous présenterons ensuite le prototype en lui-même. Nous illustrerons enfin quelques subtilités inattendues que posent les grammaires attribuées modulaires, que la réalisation du prototype a permis de mettre en lumière.

2 Grammaires attribuées modulaires

Pour définir de façon compréhensible la notion de grammaires attribuées modulaires dont il est question ici, on partira d'une notion simple de grammaires et de documents sans modularité ni attributs. On ajoutera ensuite le calcul des attributs, puis la modularité.

2.1 Grammaires

Une grammaire (ou signature multi-sortes) $G = (S, \Omega)$ est un couple formé d'un ensemble fini de sortes S et d'un ensemble fini de productions Ω. Les sortes ont un nom (par exemple "entier", "variable", "chaîne de

1. Knuth, Donald E., 'Semantics of context-free languages', *Theory of Computing Systems*, vol. 2, no. 2, 127-145 (1968).

caractères", "couple"). Les productions ont un nom (par exemple "somme", "conse") et un type (également appelé signature) appartenant à $S^* \times S$.

On représente habituellement une grammaire par un graphe orienté ayant pour nœuds les sortes (représentées par des ovales) et les productions (représentées par des rectangles), et ayant pour chaque production de type $s_1, ..., s_n \to s$ une arête entrante vers la sorte s et des arêtes sortantes vers les sortes $s_1, ..., s_n$. Notons que cela impose que les arêtes sortantes des productions soient ordonnées, car les types $s_1, s_2 \to s$ et $s_2, s_1 \to s$ sont distincts.

Pour plus de clarté, et pour des raisons que justifient la notion de modularité qui sera introduite plus tard, on étiquette fréquemment les arêtes sortantes des productions ; ces étiquettes sont appelées des sélecteurs.

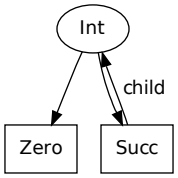

FIGURE 1 – Grammaire des entiers naturels

La figure 1 représente une grammaire décrivant les entiers naturels (avec une représentation unaire). Elle dispose d'une unique sorte "Int" ayant pour productions associées "Zero" (de type $() \to Int$) et "Succ" (de type $Int \to Int$). L'arête sortante de la production "Succ" est étiquetée par le sélecteur "child".

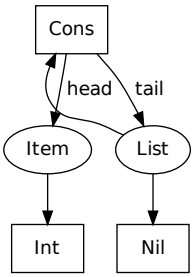

FIGURE 2 – Grammaire des listes

La figure 2 représente une grammaire décrivant les listes d'entiers. Elle dispose d'une sorte "List" représentant la liste, et une sorte "Item" représentant les entiers (d'où part une production "Int" ; on ne cherche pas à expliciter ici la construction des entiers). Les listes se construisent soit avec "Nil", soit avec "Cons" à partir de la donnée d'un élément ("head") et d'une liste ("tail").

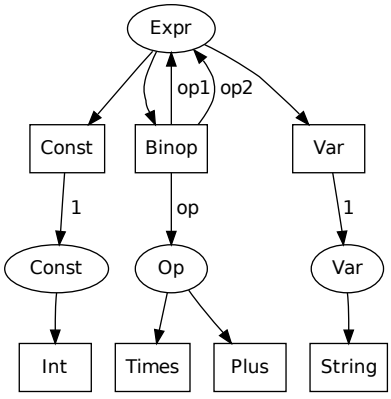

FIGURE 3 – Grammaire d'expressions mathématiques

La figure 3 représente une grammaire décrivant des expressions mathématiques simples. Une expression est soit une constante (se ramenant finalement à un entier), soit une variable (une chaîne de caractères), soit la combinaison de deux expressions par un opérateur binaire qui est soit "Plus", soit "Times".

La figure 4 représente une grammaire décrivant des arbres d'entiers. Les trois sortes "Tree", "Forest" et "Item" s'articulent de la façon suivante : un élément de sorte "Tree" est un couple formé d'une étiquette de

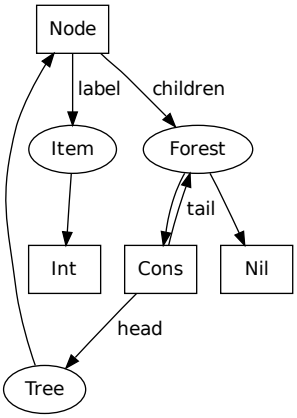

FIGURE 4 – Grammaire des arbres d'entiers

sorte "Item" (l'étiquette de sa racine) et d'un élément de sorte "Forest" qui est en fait une liste de descendants (de sorte "Tree"). Remarquons que cet exemple contient une copie de la structure de liste que nous avions définie un peu plus haut ; c'est la volonté de réutiliser des structures usuelles à plusieurs endroits de la grammaire sans en faire plusieurs copies qui nous conduira à la modularité dont il sera question par la suite.

2.2 Documents

Informellement, si les grammaires que nous venons de définir permettent de décrire des structures de données, un document conforme à une grammaire est une instance d'une telle structure. Si on pense à la représentation des grammaires sous forme de graphes, un document est en quelque sorte un parcours dans ce graphe.

Formellement, un document conforme à une grammaire $G = (S, \Omega)$ est un graphe orienté de nœuds portant une étiquette de $S \times \Omega$, et obéissant à la condition suivante : Pour tout nœud d'étiquette $(s, \omega) \in S \times \Omega$, où ω a pour type $s_1, ..., s_n \to s'$, on impose que $s = s'$, et que les fils du nœud aient pour étiquettes $(s_1, \omega_1), ..., (s_n, \omega_n)$, dans cet ordre.

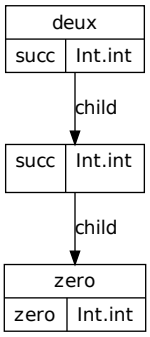

FIGURE 5 – Document pour la grammaire des entiers naturels

Exemple. La figure 5 représente un document conforme à la grammaire des entiers naturels présentée à la figure 1. On a donné ici à certains nœuds des noms spécifiques ("zero", "two") pour plus de lisibilité. La production utilisée est indiquée à gauche, et la sorte est indiquée à droite. (La partie avant le point indique le module utilisé ; cela sera expliqué au moment où nous étudierons la modularité.)

La construction d'un document peut se voir sous la forme d'un processus interactif se déroulant de la façon suivante : on choisit d'abord une sorte s pour l'étiquette d'un premier nœud. On choisit ensuite une production ω parmi celles permettant de produire cette sorte (c'est-à-dire les filles de la sorte dans la représentation de la grammaire sous forme de graphe). On crée alors les descendants du premier nœud, étiquetées par les sortes qui sont celles de la partie gauche de la signature de la production ω choisie. On continue ensuite le processus de choix d'une production sur les nouveaux nœuds ainsi créés, ce qui créera de nouveaux nœuds, et ainsi de suite.

Il n'est pas garanti que ce processus termine. Pour cette raison, on accepte que certains nœuds du document n'aient pas (encore) de production associée. Leur étiquette est alors de la forme $(s, \bot)$. De tels nœuds sont

appelés "bourgeons".

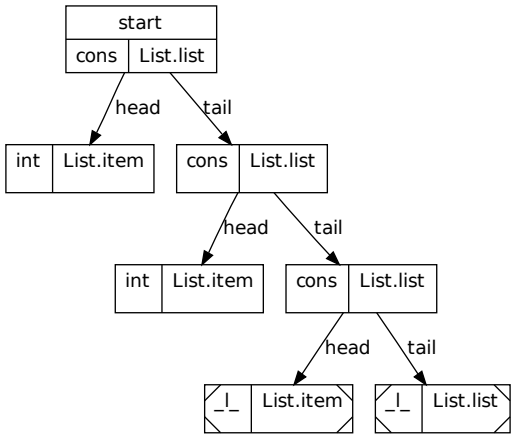

FIGURE 6 – Document pour la grammaire des listes chaînées d'entiers

Exemple. La figure 6 représente un document conforme à la grammaire des listes chaînées d'entiers naturels présentée à la figure 2. Le document n'est pas encore totalement construit, comme en témoignent les deux bourgeons que l'on aperçoit en bas de la figure.

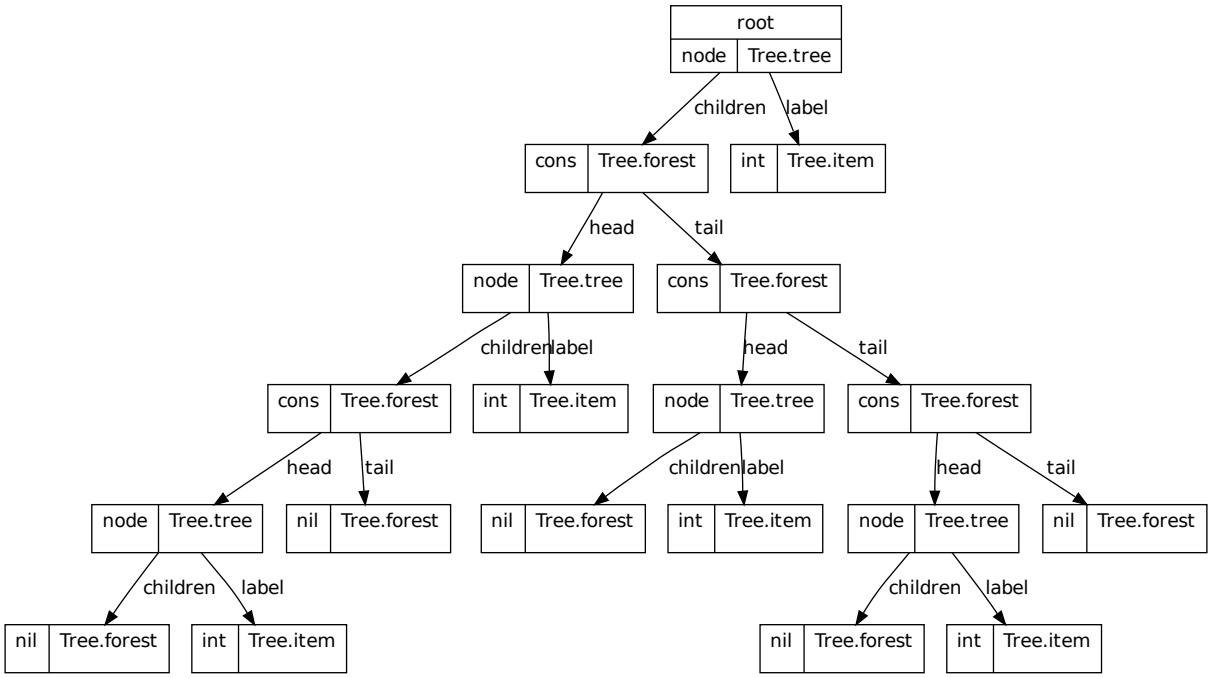

FIGURE 7 – Document pour la grammaire des arbres d'entiers

La figure 7 représente un document (un peu plus conséquent) conforme à la grammaire des arbres d'entiers naturels présentée à la figure 4. Cette fois, le document est entièrement construit et ne présente pas de bourgeons.

Notons que le processus de construction ainsi décrit ne créera que des documents ayant une forme arborescente. Pour pouvoir construire des graphes quelconques, il est nécessaire d'ajouter une opération de partage permettant de réutiliser un nœud existant du document (de sorte compatible) à l'emplacement d'un bourgeon.

La figure 8 représente un document conforme à la grammaire des listes chaînées d'entiers présentée à la figure 2. Il y a eu réutilisation du nœud "root" dans la construction du document.

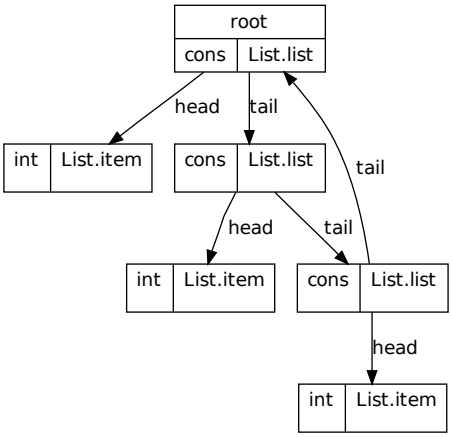

FIGURE 8 – Document circulaire pour la grammaire des listes chaînées

2.3 Attributs

2.3.1 Présentation générale

Le mécanisme d'attributs permet de définir formellement la sémantique d'une grammaire. Informellement, on créera au niveau de la grammaire des attributs et règles sémantiques, et on calculera la valeur de ces attributs dans les documents.

Pour présenter la notion d'attributs, on exposera dans un premier temps les cas plus simples des attributs purement synthétisés et purement hérités, avant de passer au cas général.

2.3.2 Attributs purement synthétisés

Dans le cas d'attributs purement synthétisés, la règle de calcul d'un attribut s'exprime directement en fonction des nœuds fils dans le document.

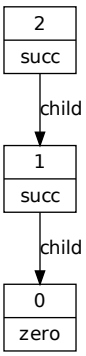

FIGURE 9 – Exemple de calcul d'attribut purement synthétisé

Si l'on définit, pour la grammaire présentée en figure 1, un attribut "value" défini par les règles sémantiques $value = 0$ pour "zero" et $value = child.value + 1$ pour "succ", on aboutit au calcul illustré par la figure 9. On définit ainsi la sémantique permettant d'associer sa valeur à un document représentant un entier. Notons que, dans le cas d'un document circulaire similaire à celui de la figure 8, la définition de l'attribut n'aurait plus eu de sens ; ce problème sera traité plus bas.

Une sémantique de ce genre est suffisante pour formaliser un bon nombre de calculs usuels : longueur d'une liste, nombre d'éléments d'un arbre, profondeur minimale ou maximale, valeur d'une expression mathématique ne comportant pas de variables, etc.

2.3.3 Attributs purement hérités

Si on considère que les attributs purement synthétisés correspondent à une remontée d'information du fils vers le père, il est naturel de s'intéresser au sens inverse, où l'information est transmise du père vers le fils.

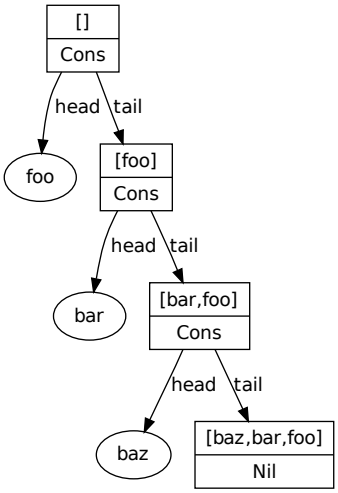

FIGURE 10 – Exemple de calcul d'attribut purement hérité

Si l'on définit, pour la grammaire présentée en figure 2, un attribut "value" défini par la règle sémantique $value = head :: father.value$ pour "cons", et initialisé à la liste vide, on aboutit au calcul illustré par la figure 10. On définit ainsi la sémantique permettant de calculer la liste miroir d'une autre liste. Remarquons que celle-ci correspond clairement à l'algorithme OCaml suivant :

```
let rev list =
  let rec rev list acc = match (list, acc) with
  | ( [] , l) -> l
  | (h::t, l) -> rev t (h::l)
  in rev list []
```

2.3.4 Définition générale

Les grammaires attribuées générales permettent de faire circuler des informations dans les deux sens. Plus formellement, on munit chaque sorte de la grammaire un ensemble d'attributs que l'on suppose partitionné en deux ensembles disjoints : les attributs hérités, et les attributs synthétisés. Puis, pour chaque production ω de la grammaire, ayant pour type $s_1, ..., s_n \to s$, on définit les attributs d'entrée et de sortie comme suit :

- Les *attributs d'entrée* sont les attributs synthétisés des $s_1, ..., s_n$ ainsi que les attributs hérités de s.
- Les *attributs de sortie* sont les attributs hérités des $s_1, ..., s_n$ ainsi que les attributs synthétisés de s.

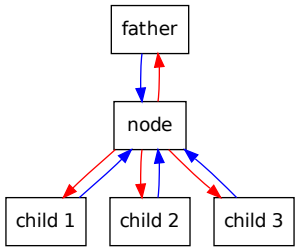

FIGURE 11 – Schéma de base pour un nœud. Les flèches allant de bas en haut (resp. de haut en bas) représentent les attributs hérités (resp. synthétisés). Les attributs d'entrée sont en bleu, les attributs de sortie sont en rouge.

Pour chaque attribut de sortie, on munit ω d'une règle sémantique permettant de calculer cet attribut de sortie en fonction des attributs d'entrée. Ce point est illustré par la figure 11.

Remarquons qu'à la différence des attributs purement hérités et purement synthétisés, il peut être nécessaire d'effectuer plusieurs passages dans le document pour calculer la valeur de certains attributs. C'est le cas, par exemple, si, pour l'un des descendants du nœud courant, un attribut hérité H dépend d'un des attributs synthétisés S_1, et un autre attribut synthétisé S_2 dépend de H : une première exploration de l'arbre enraciné en ce descendant est nécessaire pour calculer S_1, puis, connaissant H, un deuxième passage permet de calculer S_2. Un exemple est donné en figure 12.

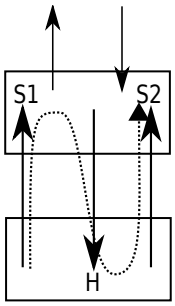

FIGURE 12 – Exemple d'un calcul d'attributs nécessitant plusieurs passes.

Remarquons également que, même si on se limite à des documents sans partage de nœuds et ayant une forme arborescente, il peut y avoir des problèmes de dépendances circulaires d'attributs avec la définition que nous avons donnée. Par exemple, il se peut qu'un nœud passe à son fils un attribut hérité dépendant d'un attribut synthétisé par ce fils, et qu'au niveau du fils, l'attribut synthétisé dépende de l'attribut hérité passé par le père. Notons que, pour une grammaire attribuée donnée, l'apparition de problèmes de ce genre peut avoir lieu ou non en fonction des choix effectués dans la construction du document. Il n'est pas simple, en général, de déterminer si, pour une grammaire attribuée donnée, il existe un document pouvant donner lieu à une dépendance circulaire entre attributs. Il existe en revanche des conditions suffisantes permettant d'éliminer ces situations à coup sûr sans perdre trop d'expressivité.

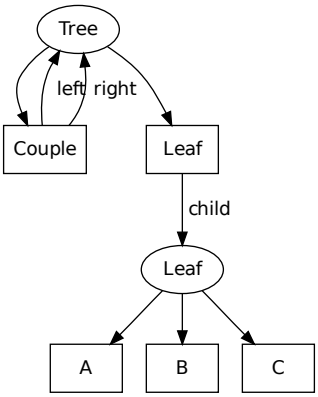

FIGURE 13 – Grammaire des arbres binaires.

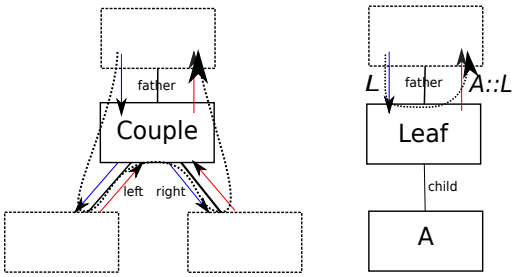

FIGURE 14 – Règles sémantiques pour la grammaire attribuée des arbres binaires permettant le calcul de la liste des feuilles.

Intéressons-nous à la grammaire représentée en figure 13 qui représente des arbres binaires. Nous souhaiterions formaliser la sémantique permettant de produire, à partir d'un arbre conforme à cette grammaire, la liste de ses feuilles. Nous munirons pour cela la sorte Tree d'un attribut hérité H et d'un attribut synthétisé S. Les règles sémantiques seront comme suit (avec des notations évidentes), avec une illustration en figure 14 :

- Pour Leaf, $father.S = child :: father.H$.
- Pour Couple :
  - $father.S = right.H$

– $right.H = left.S$
– $left.H = father.H$

Il est facile de vérifier que pour un document conforme à cette grammaire, si l'on fixe $H =$`[]` à la racine du document, S vaudra la liste des feuilles du document. Cet exemple de calcul d'attributs est effectivement implémenté par le prototype.

Si on ajoute le partage de nœuds dans les documents, on se rend compte qu'il est nécessaire d'ajouter une autre contrainte au document : un nœud du document étiqueté avec une sorte possédant des attributs hérités ne doit avoir qu'un seul père. En effet, dans le cas contraire, les différents pères pourraient fournir des valeurs différentes pour les attributs hérités, et il ne serait pas possible de savoir laquelle choisir.

2.3.5 Nature des attributs

Dans tout ce qui précède, nous n'avons pas cherché à détailler le type des attributs ou l'expressivité des règles sémantiques. Une façon simple de procéder est de considérer que les attributs sont des objets d'un type Haskell donné et que les règles sémantiques sont des fonctions Haskell prenant comme paramètres les attributs d'entrée nécessaires, et fournissant les attributs de sortie.

Cependant, une autre solution serait de dire que le type des attributs est une sorte de la grammaire, et que les règles sémantiques donnent comme valeur aux attributs de sortie un bout de document (de la bonne sorte) construit à partir des attributs d'entrée avec des productions de la grammaire. Dans des cas simples (par exemple pour le calcul de la liste miroir présenté plus haut), cette solution fonctionne très bien. Dans des cas plus complexes, on peut supposer (et la modularité rendra cette hypothèse tout à fait raisonnable) que l'on dispose dans la grammaire de sortes correspondant aux types de Haskell et à ses constructions syntaxiques. Les règles sémantiques, au lieu de produire une valeur fournie par l'évaluation d'un code Haskell, produiraient un document décrivant l'arbre de syntaxe abstraite du code Haskell à évaluer.

Cette approche peut paraître compliquée, mais elle est en fait beaucoup plus simple à implémenter, car elle signifie que les règles sémantiques ne sont que des bouts de document conformes à la grammaire (dont on suppose qu'elle contient les sortes et productions nécessaires pour décrire du code Haskell). Cela signifie également que la sémantique décrit finalement une transformation des documents de la grammaire en d'autres documents de la grammaire, ce qui a des implications théoriques intéressantes.

Remarquons que l'exemple de la liste des feuilles d'un arbre binaire donné plus haut est tout à fait conforme à cet exemple : les règles sémantiques ne décrivent que la construction d'un document dans la grammaire des listes à partir d'un document dans la grammaire des arbres binaires.

2.4 Modularité

La modularité va nous permettre de découper les grammaires en plusieurs fragments, afin de factoriser des structures usuelles et de pouvoir les réutiliser à plusieurs endroits. De tels fragments sont appelés *modules* ; les grammaires que nous avons définies jusqu'à présent sont en fait les modules, et la grammaire entière est un ensemble de tels modules.

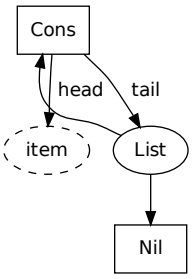

Figure 15 – Module décrivant une liste polymorphe

Présentons d'abord les choses de façon informelle avec une grammaire d'exemple. Cette grammaire contient un module List représenté en figure 15. Notons qu'aucune production ne part du nœud "item", qui est donc représenté en pointillés. Ce nœud est un *paramètre* du module ; d'autres modules de la grammaire pourront réutiliser la structure List en passant une valeur à utiliser pour "item" afin d'indiquer la sorte à utiliser pour les éléments de la liste. La grammaire contient également un module Couple représenté en figure 16, qui représente la structure de couple.

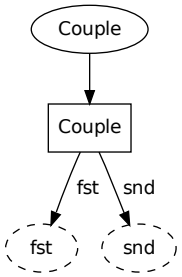

FIGURE 16 – Module décrivant un couple polymorphe

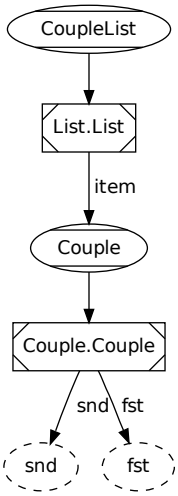

FIGURE 17 – Module décrivant une liste de couples

La grammaire contient également un module CoupleList décrivant les listes de couples, représenté en figure 17. Ce module a deux paramètres, "fst" et "snd". Il contient également les deux *définitions* CoupleList et Couple. CoupleList est défini comme étant la sorte List du module List, mais où le paramètre "item" a été remplacé par Couple ; Couple est défini à son tour comme la sorte Couple du module Couple à qui on a passé "fst" et "snd" comme paramètres.

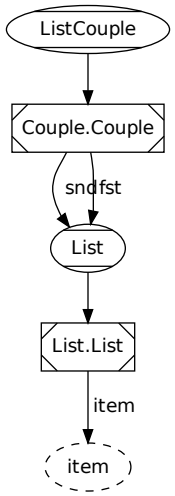

FIGURE 18 – Module décrivant un couple de listes

Terminons avec un module ListCouple décrivant les couples de listes, représenté en figure 18. Remarquons qu'il s'agit ici d'un module décrivant un couple de deux listes *de même type*.

Le graphe de dépendances entre les modules de la grammaire d'exemple que nous avons étudiée est représenté

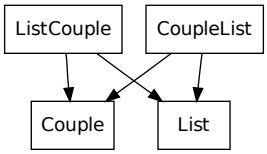

FIGURE 19 – Graphe de dépendances entre modules pour la grammaire d'exemple

en figure 19.

Formellement, la modularité s'obtient donc en construisant une grammaire modulaire à partir de modules fonctionnant comme autant de grammaires indépendantes, et par l'ajout du mécanisme de définitions. Une définition est une sorte spéciale ayant comme unique nœud fils une indirection ω, dont le nom pointe vers une sorte d'un module $M = (S, \Omega)$ externe (avec la syntaxe "module.sorte") et telle que pour tout paramètre s de M (ie. $s \in S$ tel qu'il n'existe aucun $\omega' \in \Omega$ de type de la forme $s_1, ..., s_n \to s$), ω ait un descendant avec le sélecteur s.

Notons que ces descendants peuvent également être des définitions, comme l'illustrent les exemples fournis en figures 18 et 17. En revanche, on interdit aux modules de présenter des cycles de définitions, c'est-à-dire un cycle alternant entre des productions et des indirections. (Cette situation correspondrait à une erreur "The type abbreviation is cyclic" en OCaml.)

On impose également que le graphe de dépendances ne soit pas cyclique, où le graphe de dépendances est le graphe orienté ayant pour nœuds les modules de la grammaire et ayant une arête du module A vers le module B si A contient une indirection dont le nom pointe vers B.

2.4.1 Décomposition et fusion

La modularité nous incite à considérer deux opérations naturelles :

- La décomposition, où l'on souhaite exporter un ensemble de sortes d'un module M vers un autre module M', et remplacer la définition de ces sortes par une ou des définitions dans M qui pointent vers les sortes correspondantes de M'.
- La fusion, où l'on souhaite enlever une définition et la remplacer par une copie de la structure qu'elle représente.

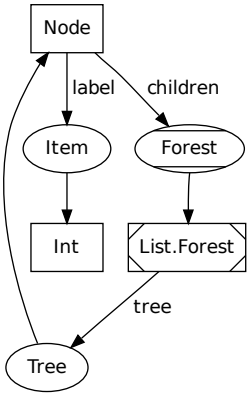

FIGURE 20 – Module des arbres après décomposition sur la sorte "Forest" vers le module List

À titre d'exemple, considérons la figure 20. Celle-ci représente le module obtenu à partir de celui de la figure 2 après décomposition de la sorte "Forest" vers le module List. Une telle décomposition nous permet de mettre la structure de liste dans un module spécifique où il sera possible de la réutiliser à plusieurs endroits sans en faire plusieurs copies. Notons que la fusion appliquée à la définition ainsi créée rétablirait exactement la situation de la figure 2, à ceci près que le module List nouvellement créé continuerait à exister sans être utilisé.

Des exemples pour la fusion sont donnés par les figures 21 et 22. Il s'agit des modules obtenus à partir des figures 18 et 17 respectivement en appliquant l'opération de fusion à toutes les définitions.

Notons que les opérations de fusion et de décomposition n'étant finalement qu'un changement de représentation de la grammaire qui n'affecte pas son sens, elles peuvent être appliquées sans que cela ne mette en danger la conformité à la grammaire des documents existants. Cependant, certains problèmes se posent malgré tout ; il en sera question ultérieurement.

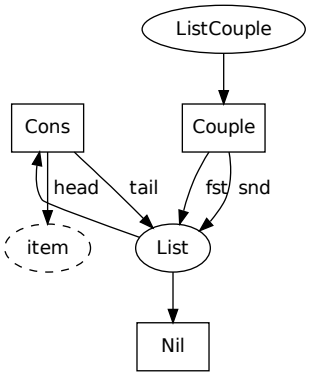

FIGURE 21 – ListCouple après fusion des définitions

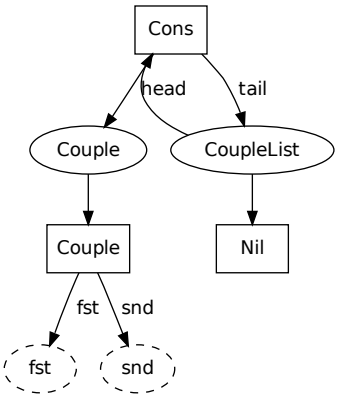

FIGURE 22 – CoupleList après fusion des définitions

2.4.2 Modularité et documents

Les documents, grâce à la modularité, peuvent comporter des sortes de plusieurs modules différents. Le passage d'un module à un autre se fait de façon transparente au moment où l'on rencontre des paramètres ou des définitions.

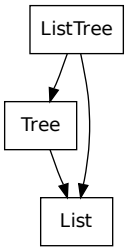

FIGURE 23 – Grammaire des arbres de listes.

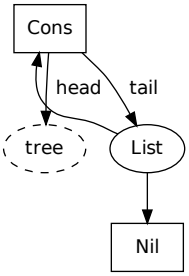

FIGURE 24 – Grammaire des arbres de listes, module des listes.

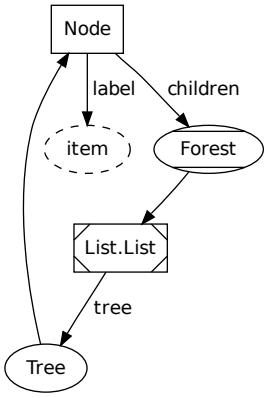

FIGURE 25 – Grammaire des arbres de listes, module des arbres.

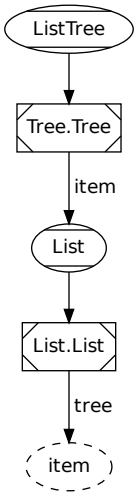

FIGURE 26 – Grammaire des arbres de listes, module des arbres de listes.

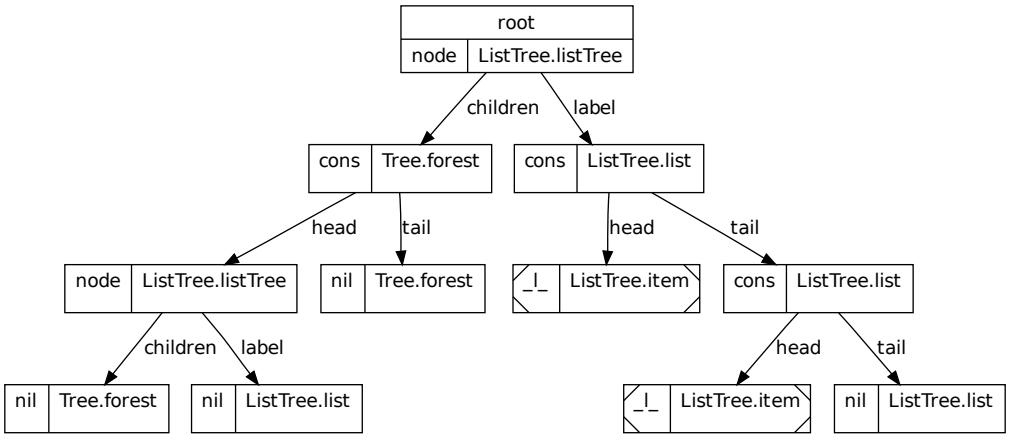

FIGURE 27 – Grammaire des arbres de listes, document d'exemple.

À titre d'exemple, considérons la grammaire des arbres de listes, dont les modules sont résumés en figure 23 et explicités en figures 24, 25 et 26. La figure 27 présente un exemple de document construit sur cette grammaire.

3 Implémentation

Le prototype réalisé pour le stage permet l'édition, avec une interface graphique, de grammaires attribuées modulaires, de documents conformes à ces grammaires, et de règles sémantiques. Les données ainsi créées sont sauvegardées sous forme de code Haskell, en prenant en charge la rédaction d'un certain nombre de fonctions et structures d'usage courant qu'il est fastidieux d'écrire à la main. Pour les modules ne comportant pas de définitions, le prototype permet de calculer la valeur d'attributs, par génération et exécution de code Haskell.

D'un point de vue technique, le prototype a été réalisé en Python, et utilise les librairies suivantes :

- PyGTK pour l'interface graphique (binding Python-GTK).
- Xdot pour le dessin des graphes (binding Python-Graphviz).
- Ply pour le parsing du code Haskell (binding Python-Lex-Yacc).

La longueur du code est de 8000 lignes environ.

3.1 Édition de grammaires

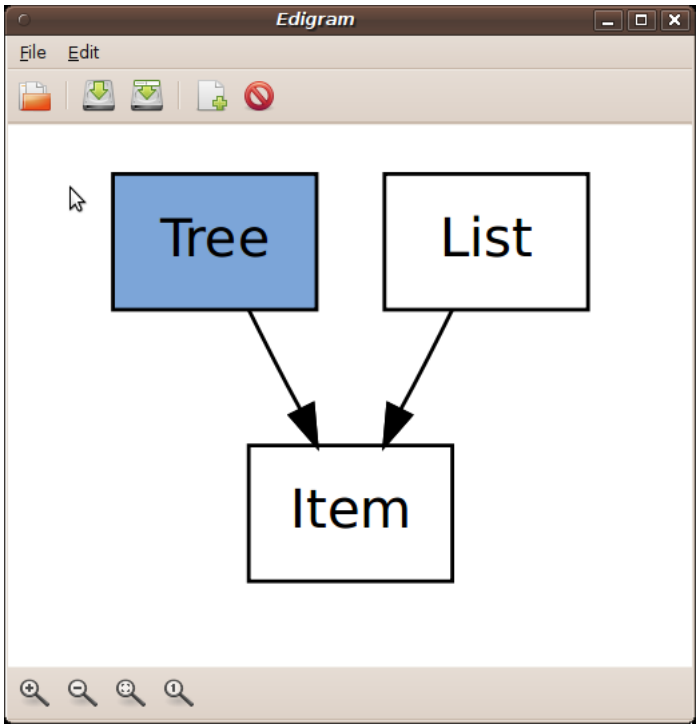

FIGURE 28 – Graphe de dépendances de la grammaire

Le prototype permet de construire des modules, et de construire les grammaires par ajout de sortes et de production et connexion des sortes et productions. La figure 28 est une capture d'écran de la fenêtre présentant le graphe des dépendances de la grammaire. La figure 29 est une capture d'écran de la fenêtre présentant le contenu du module Tree de la grammaire. Les opérations de décomposition et de fusion sont disponibles.

3.2 Édition de documents

En double-cliquant sur une sorte, on accède à une fenêtre (figure 30) présentant les nœuds du document ayant cette sorte, et permettant d'en créer de nouveaux. L'édition du document (figure 31) se fait avec une représentation de la partie du document enracinée en un nœud choisi, sous la forme d'un graphe et d'un arbre, avec la possibilité de replier certains pans du document que l'on ne souhaite pas éditer ; l'utilisateur choisit les productions à appliquer aux bourgeons, et le logiciel répond en ajoutant au document les nouveaux nœuds nécessaires. La fonction de réutilisation permet d'utiliser un même nœud du document à plusieurs endroits ; une fonction de copie est également disponible. L'utilisateur peut également revenir en arrière et faire repasser tout nœud du document à l'état de bourgeon, ce qui le déconnecte de sa descendance.

Bien entendu, une fois que des documents ont été créés, l'édition des parties correspondantes de la grammaire n'est plus possible (afin d'assurer que le document reste conforme à la grammaire. Certaines opérations restent

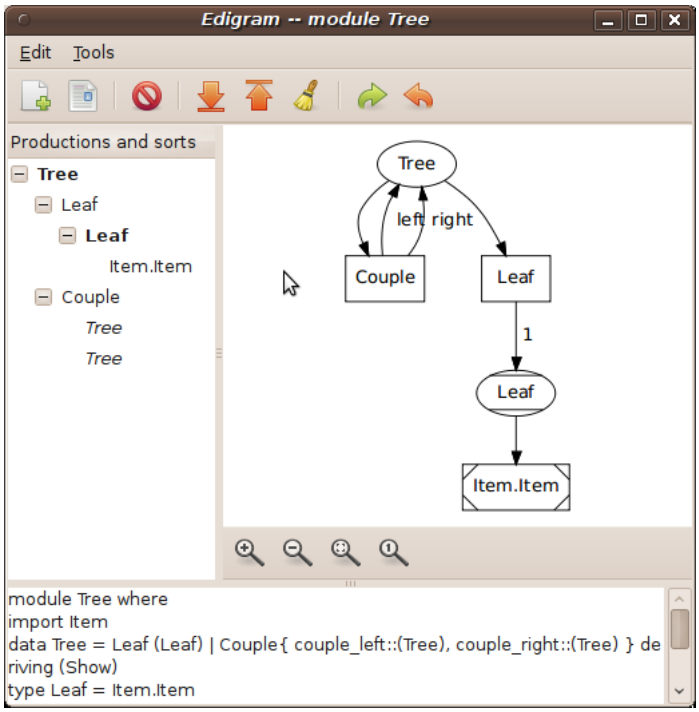

FIGURE 29 – Édition du module Tree

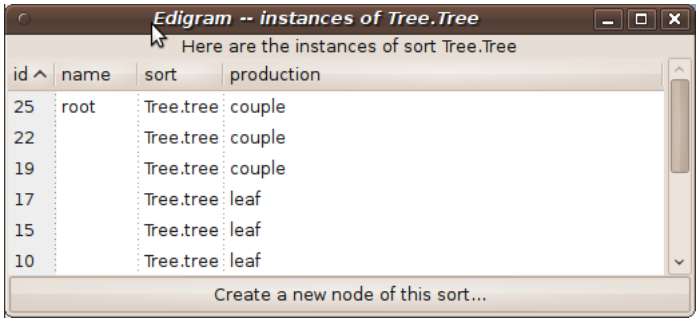

FIGURE 30 – Liste des instances de Tree.Tree

cependant autorisées (renommage de productions et de sortes, ajout de nouvelles productions, décomposition et fusion principalement).

3.3 Édition de règles sémantiques

En éditant la grammaire, l'utilisateur peut ajouter aux sortes des attributs (32), et ajouter aux productions des règles sémantiques (33). Les règles sémantiques se créent comme des documents ; l'utilisation des attributs d'entrée se fait en créant des nœuds avec des noms spéciaux.

3.4 Évaluation d'attributs

L'évaluation d'attributs se fait dans le document. L'utilisateur dispose d'une fenêtre (34) lui permettant de spécifier l'attribut à évaluer (parmi les attributs synthétisés) et de fournir des valeurs d'initialisation pour les attributs hérités du nœud sélectionné. Le prototype génère alors le code Haskell permettant d'effectuer l'évaluation, l'exécute, parse son résultat et l'affiche sous la forme d'un document.

3.5 Génération de code Haskell

Le prototype sauvegarde et lit les grammaires, documents et règles sémantiques sous la forme de code Haskell. Les grammaires sont sauvegardées comme une collection de modules Haskell contenant des déclaration

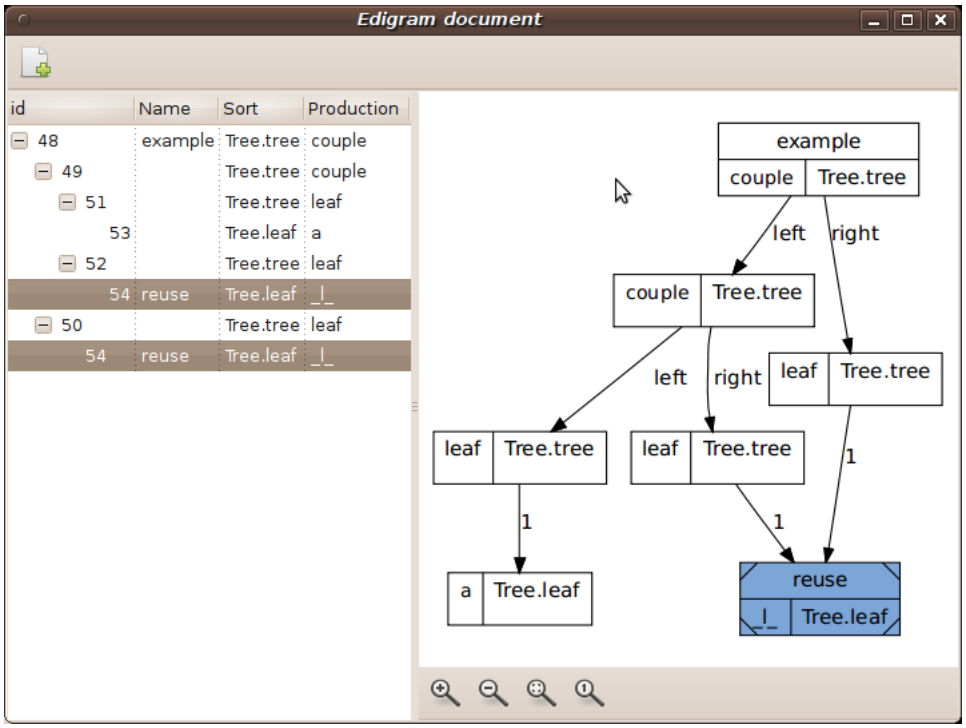

FIGURE 31 – Document d'exemple pour la grammaire

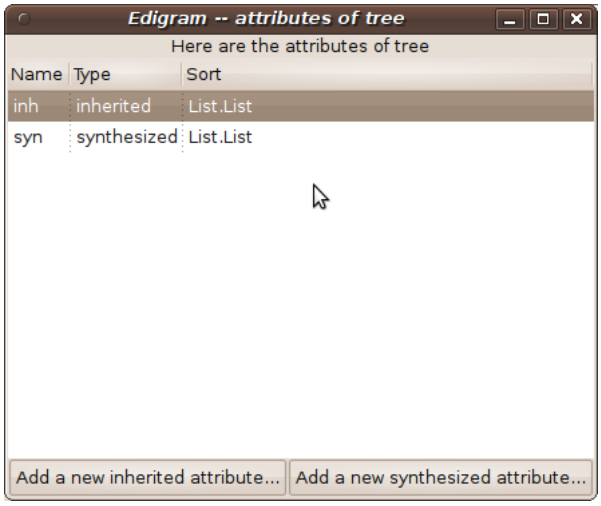

FIGURE 32 – Attributs de Tree.Tree

`data` et des déclarations `type` (pour les définitions). Les documents sont sauvegardés comme une collection de déclarations. Les règles sémantiques sont également conservées sous cette forme (elles sont gérées comme des parties de document).

Au moment de l'évaluation d'un attribut, le prototype génère le code Haskell nécessaire pour l'évaluation de l'attribut demandé. Cela inclut les règles sémantiques, les portions nécessaires du document, les modules concernés de la grammaire, ainsi que le code nécessaire pour procéder effectivement à l'évaluation d'attributs.

4 Problèmes soulevés

L'implémentation du formalisme des grammaires attribuées modulaires sous la forme d'un prototype visait également à identifier quelques points de détail pour lesquelles les choses ne se passaient pas aussi simplement que prévu. Voici quelques exemples de difficultés rencontrées.

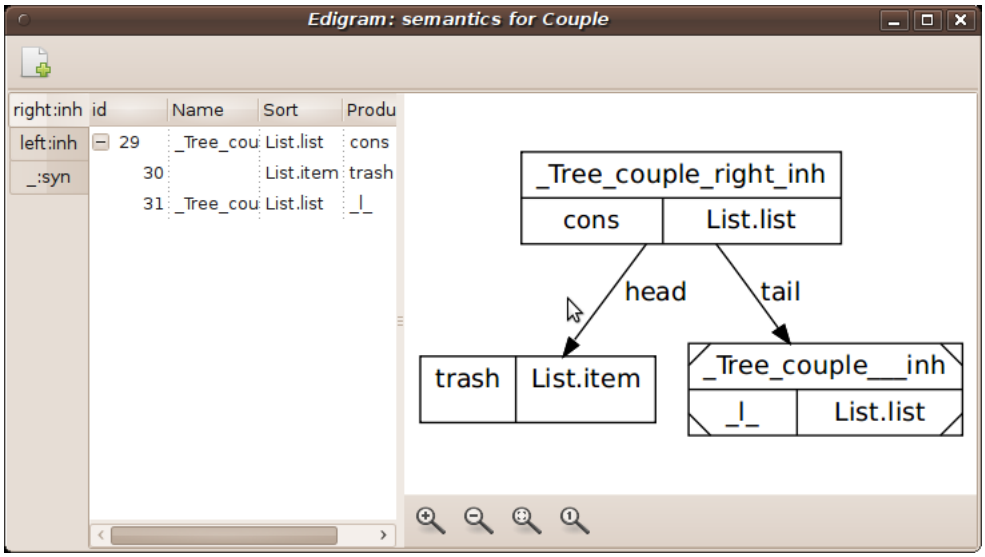

FIGURE 33 – Édition des règles sémantiques

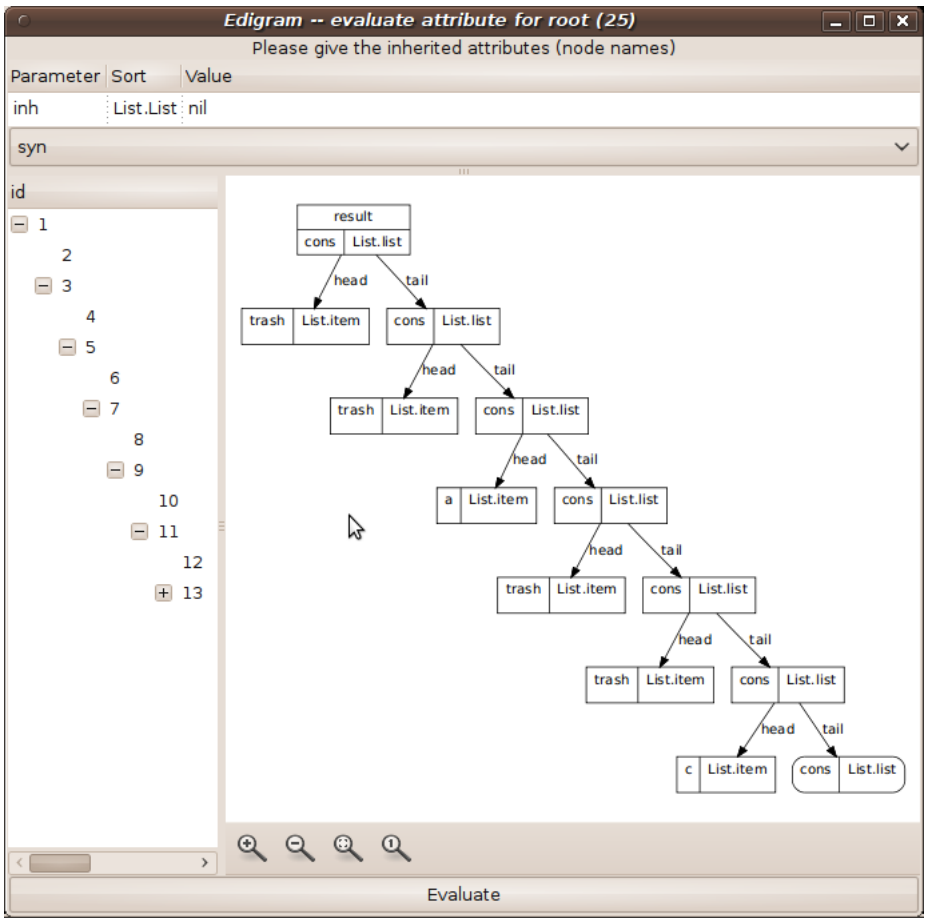

FIGURE 34 – Évaluation d'un attribut

4.1 Contextes

Le système d'étiquetage des nœuds du document que nous avons présenté s'avère insuffisant lorsqu'il s'agit de grammaires modulaires. En effet, il ne suffit plus de savoir quelle est la sorte du nœud ; il faut également mémoriser quelles définitions ont été franchies avant d'atteindre ce nœud. (Pour prendre un exemple, si List est le module des listes polymorphe, il ne suffit pas de savoir qu'un nœud est de type List.List, mais il faut

également savoir quel est le type des éléments de la liste, si celui-ci a été précisé.)

Le mécanisme retenu pour résoudre ce problème est de conserver, pour chaque nœud, un "environnement" indiquant l'historique des définitions franchies, un peu comme une pile d'appel. Lorsqu'on tente de créer un nœud ayant pour sorte une définition, on regarde vers quelle sorte cette définition pointe, et on empile la définition (et les paramètres qu'elle fournit) dans la pile d'appel. Lorsqu'on tente de créer un nœud ayant pour sorte un paramètre, on regarde si la valeur pour ce paramètre a été passées dans l'environnement. Notons que cette valeur peut à son tour être un paramètre dans le module appelant, auquel cas il faut continuer à dépiler ; ou bien, cette valeur peut être une définition, auquel cas il y aura également des informations à empiler.

Ce mécanisme pose des difficultés pratiques d'implémentation assez considérables. Du reste, les informations ainsi conservées sur l'environnement doivent être mises à jour lors des opérations de fusion et de décomposition ; le processus est complexe.

Remarquons enfin que la réutilisation de nœuds est contrainte par le contexte : pour que le document ait du sens, réutiliser un nœud à l'emplacement d'un bourgeon ne peut se faire que si le nœud à réutiliser a non seulement la sorte attendue mais aussi le contexte attendu.

4.2 Décomposition et fusion

Les opérations de décomposition et de fusion sont intuitivement simples, mais posent des problèmes inattendus. En particulier, il n'y a pas de façon simple de garantir que l'une est bien l'inverse de l'autre.

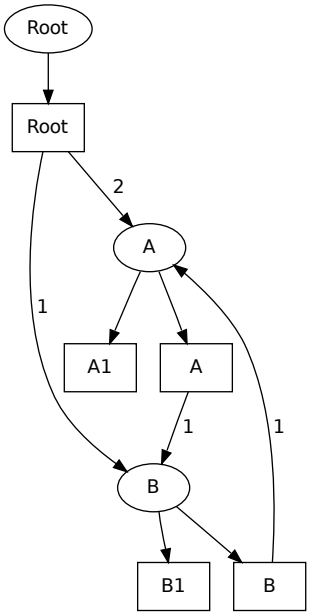

FIGURE 35 – Grammaire d'exemple, module M.

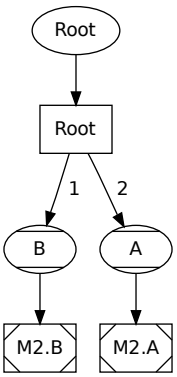

FIGURE 36 – Grammaire d'exemple après décomposition vers M2, module M.

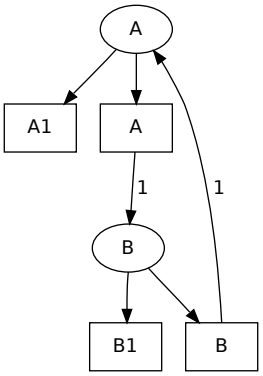

FIGURE 37 – Grammaire d'exemple après décomposition vers M2, module M2.

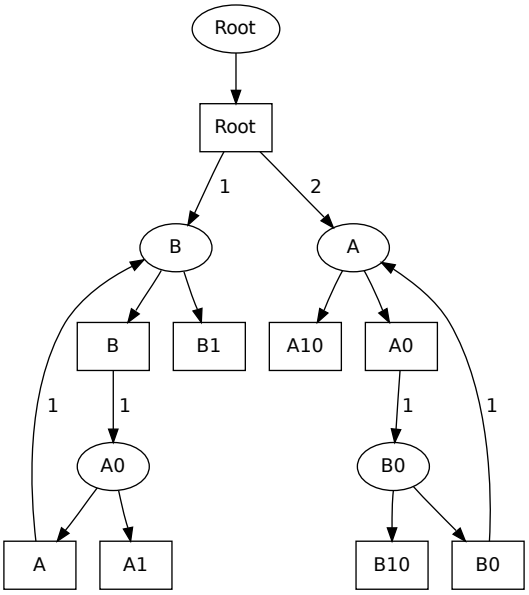

FIGURE 38 – Grammaire d'exemple après fusion, module M.

Pour s'en convaincre, considérons la grammaire ayant pour unique module le module M représenté en figure 35, et imaginons que nous souhaitons décomposer sur les deux sortes "A" et "B" pour les envoyer dans un module M2. Le résultat de cette opération est présenté en figures 36 (module M) et 37 (module M2). Le résultat semble raisonnable. Cependant, si nous effectuons la fusion des deux définitions, nous obtenons pour le module M le résultat donné en figure 38, qui n'est pas la même chose que ce que nous avions au départ (figure 35).

Il semble à première vue qu'il faudrait, au moment de la fusion, ne créer qu'une seule copie et revenir ainsi à l'état initial. C'est le cas dans cet exemple, mais cette solution simple ne fonctionne pas en général : si l'utilisateur ajoutait un paramètre au module M2 et lui passait des valeurs différentes dans les deux définitions du module M, il serait obligatoire de créer deux copies.

En fait, le problème est plus subtil encore. En effet, considérons le document de la figure 39. Il s'agit d'un document valide pour la grammaire avant décomposition (figure 35) ; cependant, après décomposition, le document n'est plus conforme à la grammaire (figure 36), puisqu'on passe subrepticement de la première définition à la deuxième.

Il s'avère donc que l'opération de décomposition n'est pas toujours légale, mais doit être interdite dans certains cas. La définition exacte des cas à interdire est assez complexe.

4.3 Déconstruction d'un document

Il semble naturel de permettre à l'utilisateur de changer d'avis lors de la construction du document, c'est-à-dire de ramener un nœud à l'état de bourgeon pour pouvoir choisir une autre production que celle qui a été sélectionnée. Cependant, il reste à décider ce que devient la descendance du nœud considéré. Il n'est pas possible

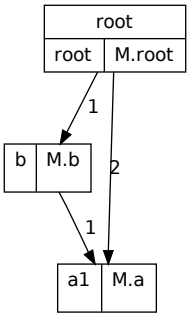

FIGURE 39 – Document pour la grammaire d'exemple

de la déconnecter sans plus de précautions, car il se pourrait qu'ainsi, elle perde des informations de contexte, et que sa descendance devienne invalide.

La solution apportée à ce problème est de réexaminer toute la descendance des nœuds dont le père vient d'être transformé en bourgeon, et de déconnecter à nouveau dès qu'il est fait usage d'un paramètre pour lequel plus aucune valeur n'est passée en amont. Le partage de nœuds, et la possibilité d'avoir des cycles dans le document, rend cette opération encore plus complexe.

5 Conclusion

La réalisation du prototype aura été l'occasion pour moi de me familiariser avec les grammaires attribuées et avec la notion de modularité proposée par mon maître de stage. Son implémentation, en plus de son caractère formateur pour ce qui est de la maîtrise pratique des outils utilisés et des bonnes pratiques de programmation, a permis d'identifier un certain nombre de points litigieux, et a donné lieu à des discussions intéressantes pour choisir la direction à suivre.

5.1 Perspectives

Le prototype peut encore être amélioré de nombreuses manières. En premier lieu, des améliorations d'interface et corrections de bugs restent possibles. De façon plus profonde, la gestion du calcul des attributs dans les cas où la modularité est effectivement exploitée pourrait être un outil utile.

Toujours dans le cadre de l'évaluation d'attributs, en suivant l'idée selon laquelle le résultat de l'évaluation est un document comportant éventuellement des nœuds représentant des mots-clés Haskell, il faudrait créer un module spécial contenant la grammaire de Haskell et dont le contenu serait transformé en du code Haskell au moment de l'évaluation.

Le développement du prototype a été confié à présent à Maurice Tchoupé Tchendji, Bernard Fotsing Talla, et Rodrigue Djeumen. L'objectif final serait la publication d'un article décrivant le formalisme des grammaires attribuées modulaires et fournissant le prototype comme implémentation.

5.2 Remerciements

Je tiens à remercier mon maître de stage, Éric Badouel, pour le temps qu'il m'a accordé, pour son aide précieuse et pour sa disponibilité sans faille.