Intégration du Filtrage Modulo dans JAVA

J.-P. Lambert[1] & P. Viry[2]

1: 4 rue des roses
91590 Cerny, France
jp.raven@worldonline.fr
2: ILOG,
9 rue de Verdun
94253 Gentilly Cedex, France
pviry@ilog.fr

Résumé

Nous présentons la conception, la sémantique et l'implantation d'une extension du langage JAVA avec des structures de données équationnelles intégrées et des instructions de filtrage modulo ces structures. Nous montrons sur quelques exemples l'intérêt d'un tel langage. Nous relatons l'expérience retirée de cet exercice, en détaillant les avantages de l'approche choisie et les problèmes rencontrés.

1. Introduction

L'extension que nous proposons à JAVA, que nous appelerons JAVAMODULO, est un langage qui a les particularités suivantes :

- Mis à part les ajouts, il se comporte comme JAVA [1], qu'il s'agisse de la syntaxe, de l'évaluation ou du typage ; de plus le comportement des ajouts est cohérent avec celui de JAVA.
- Il ajoute à JAVA de nouvelles structures de données équationnelles et permet de faire du filtrage modulo la théorie équationnelle correspondante sur ces structures.
- Il est implanté sous la forme d'un traducteur source-à-source, dont le code en sortie est du pur JAVA.

L'intérêt de ce travail est multiple :

- Il montre l'intérêt d'avoir des structures de données intégrées dans un langage avec la possibilité de filtrer sur ces structures modulo leur propriétés algébriques.
- C'est une instance du problème général de l'extension de langages, qui nous permet sur un exemple non trivial d'en estimer les possibilités et les problèmes.
- Il propose une approche pragmatique permettant de mettre entre les mains du plus grand nombre des concepts prometteurs tels que le filtrage modulo, qu'on ne rencontre encore que dans des systèmes ou langages académiques.

1.1. Structures de données équationnelles

Définition 1 (Théorie équationelle) *Soit E un ensemble d'équations $s = t$, la* théorie équationnelle engendrée par *E est la fermeture refléxive transitive de la relation engendrée par contexte et substitution par les équations de E, notée $=_E$.*

Les classes d'équivalence modulo cette relation sont notées $[t]_E$, autrement dit on a

$$[u]_E = [v]_E \quad ssi. \quad u =_E v$$

JAVAMODULO intègre à JAVA les collections suivantes, chacune correspondant à une théorie équationnelle :

- les Séquences (théorie associative-unitaire, AU),
- les Multi-ensembles (théorie associative-unitaire-commutative, AUC),
- les Ensembles Ordonnés[1](théorie associative-unitaire-idempotente, AUI), et
- les Ensembles (théorie associative-unitaire-commutative-idempotente, AUCI).

Nous nous intéressons à ces théories équationnelles pour plusieurs raisons :

- Une théorie associative unitaire forme un monoïde, indispensable pour « agréger » des données, c'est-à-dire former une collection de données ;
- la commutativité et l'idempotence forment des structures équationnelles rencontrées très souvent en informatique et sur lesquelles le filtrage équationnel est faisable efficacement pour de nombreux cas pratiques.

Intégrer des structures de données dans le langage permet une utilisation plus naturelle qu'en utilisant des structures de données externes qu'on utilise explicitement via des API. Associé au filtrage modulo qui respecte la théorie équationnelle correspondant à la structure de données (par exemple associatif-unitaire pour les Séquences), nous obtenons un langage avec une interprétation unique de ces structures de données. Nous avons ainsi une *abstraction de l'implantation.*

Mais abstraire des structures de données a aussi un défaut : avec une structure de données abstraite, on n'a plus de contrôle direct sur l'implantation, ce qui rend plus difficile voire impossible les évaluations de performance et de complexité des algorithmes sur ces structures de données.

1.2. Filtrage modulo

Le filtrage modulo permet de manipuler des structures de données équationnelles sous une forme abstraite qui correspond à la théorie équationnelle correspondante.

Définition 2 *Soit E une théorie équationnelle. On dit qu'un terme t* filtre modulo E *un terme s, et on note $t \ll_E s$, s'il existe une substitution σ telle que $s =_E \sigma t$. t est appelé* pattern *ou* motif*, s est appelé le* sujet *du filtrage et σ est appelé* filtre*.*

Le lien entre filtrage modulo et réécriture modulo est décrit dans [12].

De nombreux auteurs ont participé à l'élaboration d'algorithmes ou à prouver que l'unification et le filtrage équationnels sont des problèmes difficiles ; un panorama intéressant des complexités des problèmes d'unification et de filtrage selon les théories équationnelles utilisées est dressé par Kapur et Narendran dans [13]. Nous en retiendrons que le filtrage modulo les théories équationnelles qui nous intéressent (AU, ACU, AUI et ACUI) est NP-complet dans le cas général, mais raisonnablement efficace en pratique pour les motifs rencontrés généralement dans le contexte de la programmation.

D'un point de vue pratique, le filtrage modulo permet d'exprimer de façon abstraite et concise les algorithmes manipulant des structures de données équationnelles. En contrepartie, le programmeur perd le contrôle de la complexité des algorithmes, sauf à être aidé par des outils spécifiques (par exemple analyse de la forme des motifs).

On peut trouver des implantations du filtrage modulo dans un certain nombre de systèmes :

- les deux principaux moteurs de la réécriture : ELAN [4, 5] et MAUDE [6, 7] ;
- le projet MGS [8, 9], un langage dédié à la modélisation et la simulation de systèmes dynamiques avec une structure dynamique, permet du filtrage équationnel sur les structures topologiques de MGS ;
- TOM [10, 11] ajoute du filtrage à d'autres langages sous la forme d'un préprocesseur ; TOM permet aussi du filtrage associatif.

[1] À ne confondre avec la notion mathématique d'*ensemble ordonné* composé du couple ensemble et ordre sur cet ensemble

A notre connaissance il n'existe aucune implantation du filtrage modulo pour des langages de programmation généralistes.

2. Choix de conception

2.1. Choix d'une syntaxe et d'une sémantique

La syntaxe a été choisie pour être cohérente et naturelle pour un utilisateur JAVA. La sémantique a elle aussi été choisie pour rester cohérente avec JAVA et pour pouvoir être exprimée facilement, en rapportant directement la sémantique du filtrage à la définition du filtrage modulo une théorie équationnelle. La section 3 donne la syntaxe et la sémantique du nouveau langage.

2.2. Transformation de programmes source-à-source

Nous avons choisi d'implanter ce nouveau langage par une transformation source-à-source, de telle sorte que le code en sortie soit compilable par le compilateur JAVA standard.

Cette méthode d'implantation permet un développement plus rapide qu'en développant un compilateur complet quand on utilise un cadre de travail comme POLYGLOT (présenté plus loin) qui évite de partir de zéro. En se basant sur un autre compilateur déjà mature, les performances sont meilleures qu'avec un nouveau compilateur sans optimisation. Enfin un programme compilé avec un traducteur source-à-source peut être mis à jour même si le traducteur n'est plus maintenu : on peut travailler sur les sources en sortie du traducteur source-à-source.

Notons que cette méthode d'implantation dispose aussi de certains inconvénients. On ne peut pas apporter d'optimisation au code exécutable, qui est produit par un compilateur tierce et on doit se soumettre aux limitations de ce dernier. Le temps de compilation d'un programme est plus grand qu'avec un compilateur dédié et optimisé, puisqu'on ajoute des passes supplémentaires au processus de compilation au lieu d'étendre les passes existantes.

2.3. TOM

Au niveau du choix de l'algorithme de filtrage, les algorithmes sont nombreux, mais leur implantation est une tâche très difficile. Notre travail ne se focalise pas sur l'implantation d'un tel algorithme mais plutôt sur l'intégration du filtrage modulo dans JAVA. L'idée la plus naturelle était donc d'utiliser une implantation déjà existante. Nous utilisons donc TOM en tant que moteur de filtrage, comme une passe supplémentaire de compilation. Nous ne présentons pas ici TOM en détail ; pour plus d'information sur TOM, consulter [10, 11].

TOM est un pré-processeur de langage et est disponible pour plusieurs langages. TOM n'est pas une extension de langage ; c'est un langage de filtrage, qui propose de nouvelles fonctionnalités sans les intégrer dans le langage hôte. Nous ne produisons donc pas directement du code JAVA mais du code JAVA agrémenté de constructions TOM, ensuite compilé par TOM pour obtenir du code JAVA pur. La figure 1 montre le fonctionnement global du compilateur JAVAMODULO :

1. il transforme un fichier source JAVAMODULO en fichier source JAVA avec des constructions TOM,
2. le fichier JAVA avec des constructions TOM est transformé en un fichier source JAVA par TOM, et
3. le fichier JAVA est compilé par le compilateur JAVA pour produire un fichier binaire exécutable par la machine virtuelle JAVA.

TOM et JAVAMODULO ne proposent pas la même approche. TOM est un pré-processeur alors que JAVAMODULO est une extension de langage. Du point de vue de l'analyse du code tapé par l'utilisateur,

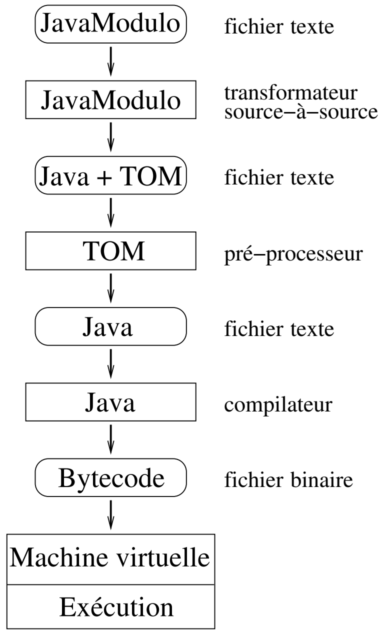

FIG. 1 – schéma de compilation et d'exécution d'un fichier source JAVAMODULO

TOM ignore tout du langage hôte et n'analyse que le code TOM alors que JAVAMODULO analyse tout le code JAVAMODULO (c'est-à-dire le code JAVA étendu par les ajouts propres à JAVAMODULO) et effectue toutes les vérifications que ferait un compilateur JAVA au travers d'une interface unique. Du point de vue de l'intégration, TOM n'est pas intégré dans le langage hôte JAVA, il s'agit clairement d'un second langage, alors que JAVAMODULO est un JAVA étendu, un JAVA avec de nouveaux types, de nouvelles valeurs et de nouvelles instructions qui sont intégrées.

Le but de JAVAMODULO n'est pas de proposer une interface JAVA aux constructions de TOM ; le but de JAVAMODULO est de proposer une extension de JAVA qui présente de manière intégrée des structures de données équationnelles et la possibilité de faire du filtrage modulo sur ces structures de données. C'est pourquoi JAVAMODULO ne donne pas accès à toutes les possibilités de TOM ; en particulier TOM permet à l'utilisateur de définir ses propres structures de données sur lesquelles filtrer, contrairement à JAVAMODULO qui impose l'utilisation de certaines structures de données. C'est un choix de conception.

Dans sa version actuelle TOM ne propose que le filtrage associatif (séquences), le filtrage associatif-commutatif étant prévu pour une version ultérieure. En conséquence la première version de JAVAMODULO ne permet le filtrage que sur les séquences.

2.4. POLYGLOT

Le transformateur source-à-source est implanté en JAVA en utilisant POLYGLOT pour faciliter le développement de l'extension. Pour plus d'information sur POLYGLOT, consulter [3, 2].

POLYGLOT propose de développer des extensions JAVA de manière non-destructive. Habituellement, l'extension d'un langage est faite de manière *ad hoc*, généralement en modifiant les sources du compilateur du langage étendu. POLYGLOT met à disposition plusieurs méthodes pour éviter une telle approche, par exemple un parseur extensible. POLYGLOT permet aussi d'étendre les ASA[2] en modifiant ou en ajoutant uniquement le nécessaire. POLYGLOT reprend toutes les étapes de la compilation pour permettre de développer un compilateur complet : analyse syntaxique (*parsing*), typage, analyse statique, vérifications de consistance du code.

POLYGLOT nous permet de développer rapidement un compilateur pleinement fonctionnel, en ne repartant pas de rien.

3. Syntaxe et sémantique

Dans cette section nous présentons la syntaxe de JAVAMODULO et la sémantique que nous lui donnons. Le résultat effectif prend la forme d'un compilateur dont nous décrivons l'implantation en section 5. Puisque JAVAMODULO est une extension de JAVA, la sémantique de JAVAMODULO est donnée de deux manières différentes mais complémentaires :

1. d'une part, en langage naturel, à l'instar de la description de JAVA donnée en [1], et
2. d'autre part, de manière plus formelle afin de pouvoir se ramener à la définition du filtrage.

Lorsque nous donnerons la description en langage naturel avec la syntaxe, nous essaierons tant que possible de rester dans l'esprit de *Java Language Specification* [1]. C'est aussi cette référence qu'il faut consulter pour obtenir des détails sur le reste de la sémantique de JAVA, qui n'est pas donnée ici.

Nous ne donnerons que la syntaxe et la sémantique des Séquences : la syntaxe et la sémantique des Multi-ensembles, Ensembles Ordonnés et Ensembles sont similaires à celles des Séquences. De plus, seules les Séquences sont implantées dans la version actuelle du compilateur de JAVAMODULO.

3.1. Définitions

Voici les équations qui permettent d'engendrer les théories équationnelles qui nous intéressent :

$$\begin{array}{lll} \text{élément unitaire} & U(+,\emptyset) & = & \{x+\emptyset = x, \emptyset + x = x\} \\ \text{associativité} & A(+) & = & \{(x+y)+z = x+(y+z)\} \\ \text{commutativité} & C(+) & = & \{x+y = y+x\} \\ \text{idempotence} & I(+) & = & \{x+x = x\} \end{array}$$

Ces théories correspondent chacune à un type de collection, avec en JAVAMODULO une notation pour les valeurs de ce type et pour le type lui-même.

type	théorie équationelle			notation des éléments	notation du type
Séquences	$AU(+,\emptyset)$	$=$	$U(+,\emptyset) \cup A(+)$	`[| ... |]`	`seq`
Multi-ensembles	$AUC(+,\emptyset)$	$=$	$U(+,\emptyset) \cup A(+) \cup C(+)$	`(| ... |)`	`mset`
Ensembles Ordonnés	$AUI(+,\emptyset)$	$=$	$U(+,\emptyset) \cup A(+) \cup I(+)$	`<| ... |>`	`oset`
Ensembles	$AUCI(+,\emptyset)$	$=$	$U(+,\emptyset) \cup A(+) \cup C(+) \cup I(+)$	`{| ... |}`	`set`

3.2. Nouveaux types

JAVAMODULO introduit quatre nouveaux types paramétrés par un autre type. Il ne s'agit pas de types polymorphes.

[2]Arbre de syntaxe abstraite, *abstract syntax tree* (AST) en anglais

```
type                      ::=  ... (les types pur Java)
                               type_Sequence
                               type_Multi_ensemble
                               type_Ensemble_Ordonne
                               type_Ensemble

type_Sequence             ::=  seq[ type ]
type_Multi_Ensemble       ::=  mset[ type ]
type_Ensemble_Ordonne     ::=  oset[ type ]
type_Ensemble             ::=  set[ type ]
```

3.3. Expressions

En JavaModulo des collections peuvent être construites directement, comme des valeurs primitives. Une collection construite directement en partie droite d'une déclaration de variable sera typée avec le type donné dans cette déclaration ; les éléments de la collection devront avoir un type compatible avec ce type. Dans le cas d'une collection construite en dehors d'une déclaration, le type du premier élément de la collection sera inféré et la collection sera typée avec ce type ; les autres éléments de la collection devront avoir un type compatible avec ce type. Cela peut parfois créer des problèmes, dans ces cas-là il faut déclarer explicitement le type de la séquence dans une déclaration de variable afin de donner à la collection le type dont tous les éléments sont sous-types. Une collection vide n'est acceptée qu'en partie droite de déclaration de variable. La syntaxe de ces nouvelles valeurs suit la grammaire BNF suivante :

```
expression          ::=  ... (les expressions pur Java)
                         Sequence
                         Multi_ensemble
                         Ensemble_Ordonne
                         Ensemble

Sequence            ::=  [| |]  |  [| initialisations |]
Multi_ensemble      ::=  (| |)  |  (| initialisations |)
Ensemble_Ordonne    ::=  <| |>  |  <| initialisations |>
Ensemble            ::=  {| |}  |  {| initialisations |}

initialisations     ::=  initialisation  |  initialisation , initialisations
```

initialisation représente une partie droite de déclaration de variable, c'est-à-dire une expression ou un initialisateur de *array*.

De plus, les opérateurs + et += sont surchargés de façon à permettre la composition de collections :

```
seq[T] + seq[T]     ->  seq[T]
mset[T] + mset[T]   ->  mset[T]
oset[T] + oset[T]   ->  oset[T]
set[T] + set[T]     ->  seq[T]

seq[T] += T
mset[T] += T
oset[T] += T
set[T] += T
```

L'opérateur Java + peut servir à concaténer des collections. Dans ce cas les deux opérandes doivent être des collections de même type. L'opérateur JavaModulo + a donc trois utilisations : c'est un opérateur d'addition sur les entiers et flottants, un opérateur de concaténation sur les chaînes de caractères Java `String` ou un opérateur de concaténation sur les collections.

L'opérateur d'affectation JAVA `+=` peut servir à ajouter un élément à une collection. L'opérande gauche doit être une collection. L'opérande droite doit être de même type que les éléments de cette collection. L'opérateur d'affectation JAVAMODULO `+=` a donc trois utilisations : il permet de modifier la valeur d'un entier ou d'un flottant, d'ajouter une chaîne de caractères JAVA `String` à une autre chaîne de caractères ou d'ajouter un élément à une collection.

Il existe une bijection triviale entre les collections de JAVAMODULO et les termes modulo la théorie correspondante, qui étend la bijection entre expressions Java et termes algébriques, que nous noterons $*(\_)$.

$$
\begin{array}{lcl}
\texttt{*(\_)} & : & \textit{expression} \texttt{ -> } T_{AU} \\
\\
\texttt{*([| |])} & \texttt{->} & \emptyset_{AU} \\
\texttt{*([|}\ e_1\texttt{, ..., }e_n\ \texttt{|])} & \texttt{->} & \texttt{*}(e_1) +_{AU} ... +_{AU} \texttt{*}(e_n) \\
\texttt{*}(c_1 \texttt{ + } c_2) & \texttt{->} & \texttt{*}(c_1) +_{AU} \texttt{*}(c_2)
\end{array}
$$

Et de même pour les autres types de collection avec la théorie équationnelle correspondante.

Si S et T sont des collections JAVAMODULO de même type, l'expression S `+` T s'évalue en U tel que $*(S) + *(T) = *(U)$.

Si S est une collection JAVAMODULO et x est une expression JAVAMODULO de même type que les éléments de S, l'affectation JAVA S `+=` x s'évalue en S `=` T tel que $*(S) + [x] = *(T)$.

3.4. Instructions

JAVAMODULO propose deux nouvelles instructions : les constructions `match` et `matchall`.

Les syntaxes de ces deux instructions sont similaires. La syntaxe de la construction `match` suit la grammaire *match* et celle de la construction `matchall` la grammaire *matchall* :

match ::= **match** (*expression*) { *motif_et_action** }

matchall ::= **matchall** (*expression*) { *motif_et_action** }

motif_et_action ::= *motif* -> *statement*
 | *motif* **if** *expression_booleenne* -> *statement*

motif ::= *motif_simple* | *motif_etendu*

motif_simple ::= *expression* | ?*identificateur* | ??*identificateur* | _ | ... | [| *motifs* |]

motif_etendu ::= _ (*etiquetages*)
 | ?*identificateur* (*etiquetages*) | *nom_de_classe* (*etiquetages*)
 | ?*identificateur* : *nom_de_classe* (*etiquetages*)

motifs ::= *motif* | *motif* , *motifs*

etiquetages ::= | *etiquetage* | *etiquetage* , *etiquetages*

etiquetage ::= *identificateur* : *appel_anonyme_de_methode*

expression représente une expression JAVAMODULO. *expression_booleenne* représente une expression JAVAMODULO qui s'évalue en une constante booléenne qui est soit `true` soit `false`. *statement* correspond à une instruction JAVAMODULO. *identificateur* représente un identificateur JAVA valide. *nom_de_classe* représente un nom de classe JAVA. *appel_anonyme_de_methode* représente un appel de méthode JAVA dont l'objet cible de l'appel n'est pas donné ; par exemple `toString()` est un appel de méthode anonyme alors que `o.toString()` est un appel de méthode qui n'est pas anonyme.

Notons les deux sortes de motifs possibles : les motifs simples, qui correspondent à la grammaire *motif_simple*, qui sont des motifs sur lesquels la définition mathématique du filtrage s'applique directement, et les motifs étendus, qui correspondent à la grammaire *motif_etendu*, qui sont des motifs munis des extensions propres à JAVAMODULO, afin de pouvoir filtrer non seulement sur la structure des termes mais aussi sur leur type et sur la valeur des champs et du résultat de l'évaluation des méthodes des objets.

Le motf `?x :Integer()` filtrera sur un objet de type `Integer` et instanciera cet objet par la variable `x`. Le motif `Integer(val :longValue())` filtrera sur un objet de type `Integer` et mettra à disposition une nouvelle variable `val` qui contiendra le résultat de l'appel de méthode `longValue()` appliqué à cet objet.

La section suivante décrit de façon détaillé la sémantique de ces instructions.

4. Sémantique des instructions de filtrage

On utilise les motifs de la forme ?*identificateur* ou ??*identificateur* pour introduire de nouvelles variables instanciées par filtrage :

- Le motif *expression* doit être une expression JavaModulo valide : aucune nouvelle variable ne peut y être introduite par filtrage.
- Le motif `?x` sert à instancier un terme ; dans un motif de collection `x` est défini comme un élément de cette collection. Si `x` est déjà défini dans le contexte, alors une nouvelle définition de `x` instanciée par filtrage remplace celle du contexte, sa portée s'étend jusqu'à la fin de la construction `match` ou `matchall`.
- Le motif `??x` sert à instancier une collection de termes ; dans un motif de collection `x` est défini comme une sous-collection de cette collection, pouvant contenir un nombre arbitraire d'éléments. Si `x` est déjà défini dans le contexte, alors une nouvelle définition de `x` instanciée par filtrage remplace celle du contexte, sa portée s'étend jusqu'à la fin de la construction `match` ou `matchall`.

Les motifs `_` et `...` sont les motifs anonymes (ou *jokers*) correspondant respectivement aux motifs `?x` (*joker d'élément*) et `??x` (*joker de liste*).

Les motifs étendus permettent de filtrer *sur le type* des termes ainsi que récupérer de manière pratique les valeurs des champs et des résultats d'évaluation des méthodes des termes filtrés, par une sorte *d'étiquetage* :

1. Un motif de la forme `t(`l`)` ou `?x:t(`l`)` pose la condition que, après filtrage, le terme qui instancie ce motif soit de type `t`, de telle sorte que si nous notons $\bullet$ ce terme alors l'expression booléenne $\bullet$ `instanceof t` s'évalue en `true`.
2. Un motif de la forme `_(`l`)`, `t(`l`)`, `?x(`l`)` ou `?x:t(`l`)` avec $l = x_1 : m_1$, ..., $x_n : m_n$ pose que x_i doit être initialisé par le résultat de l'application de m_i au terme qui instancie le motif, de telle sorte que si nous nommons $\bullet$ ce terme alors $x_i = \bullet.m_i$
3. Enfin les motifs de la forme `?x(`l`)` ou `?x:t(`l`)` permettent aussi de donner un nom de variable au terme qui instancie le motif, comme on le ferait avec le motif `?x`.

Les motifs JavaModulo doivent être linéaires et ne pas instancier une variable déjà définie dans le contexte qui serait également utilisée dans le motif en tant que constante. Cependant, le compilateur JavaModulo ne vérifie pas si cette contrainte est respectée par l'utilisateur. Nous considérerons dans la sémantique que ce cas de figure ne se produit jamais.

Dans une construction `matchall`, aucune action ne doit modifier le sujet du filtrage. Cependant, le compilateur JavaModulo ne sait pas vérifier si cette contrainte est respectée par l'utilisateur.

4.1. Sémantique formelle

La définition du filtrage ne s'applique qu'à des termes ; nous avons besoin de fonctions pour convertir les termes JavaModulo et les motifs JavaModulo en termes.

Soient X l'ensemble infini des noms de variables Java, I l'ensemble des instructions JavaModulo, E l'ensemble des expressions de JavaModulo, B l'ensemble des expressions booléennes JAVAMODULO, P l'ensemble des motifs de JAVAMODULO et T l'ensemble des termes.

Définition 3 (Contexte) *Un* contexte C *est un ensemble de couples* $(x,e) \in X \times E$ *où* $\nexists x$ *tel que* $(x,e_1) \in C$ *et* $(x,e_2) \in C$ *avec* $e_1 \neq e_2$*. On dit que la variable* x *est* définie *dans le contexte* C *si* $\exists e$ *tel que* $(x,e) \in C$*. Soit* C *l'ensemble des contextes.*

Définition 4 (Contexte de filtrage) *Soient* $? \notin E$, $?? \notin E$ *et* $E^+ = E \cup \{?,??\}$*. Un* contexte de filtrage C *est un ensemble de couples* $(x,e) \in X \times E^+$ *où* $\nexists x$ *tel que* $(x,e_1) \in C$ *et* $(x,e_2) \in C$ *avec* $e_1 \neq e_2$*. On dit que la variable* x *est* définie *dans le contexte de filtrage* C *si* $\exists e$ *tel que* $(x,e) \in C$*. Lorsque* $(x,?) \in C$*, on dit que* x est une variable d'élément introduite par filtrage*. Lorsque* $(x,??) \in C$*, on dit que* x est une variable de liste introduite par filtrage*. Soit* C^+ *l'ensemble des contextes de filtrage. Notons que* $\mathsf{C} \subset \mathsf{C}^+$*, c'est-à-dire que tout contexte est aussi un contexte de filtrage.*

Nous avons besoin d'un opérateur qui permette de fusionner plusieurs contextes. Une simple union ensembliste des contextes introduirait plusieurs valeurs pour la même variable lorsqu'une variable définie dans l'environnement est également introduite par filtrage. C'est pourquoi nous définissons l'opérateur d'*union de contextes* qui assure que lorsqu'une variable est introduite par filtrage, sa définition dans l'environnement est absente.

Définition 5 (Union de contextes) *Soit* $\cup_c$ *tel que* $A \cup_c B = (A \cup B) - C$ *avec* $C = \{(x,e) | e \neq ?, (x,e) \in (A \cup B), (x,?) \in (A \cup B)\} \cup \{(x,e) | e \neq ??, (x,e) \in (A \cup B), (x,??) \in (A \cup B)\}$*. On appelle* $\cup_c$ *l'opérateur d'*union de contextes*.*

Définition 6 (Instanciation de contexte) *Soient une substitution* $\sigma = \{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}$ *et un contexte de filtrage* C^+*. On dit que* C instancie le contexte *de filtrage* C^+ *par* σ *lorsque* $C = (C^+ - \{(x,e) | (x,e) \in C^+, x \in \{x_1, \ldots, x_n\}\}) \cup \{(x_1,s_1), \ldots, (x_n,s_n)\}$

La conversion des termes JAVAMODULO vers les termes est faite par la fonction $*(\_)$ définie plus haut. $Conv$ traduit les motifs étendus en termes avec une expression booléenne qui correspond aux filtrages sur le type des motifs et une suite d'instructions qui servira à déclarer et instancier les étiquetages.

$Conv$ prend aussi en argument et retourne un contexte de filtrage. Le sens du contexte de filtrage C est le suivant :

- si $(\mathtt{x},e) \in C$ avec $e \neq ?$ et $e \neq ??$ alors `x` est une variable définie dans le contexte et elle peut être considérée comme une constante pour les besoins du filtrage ;
- si $(\mathtt{x},?) \in C$ alors `x` n'est pas une variable définie dans le contexte mais sera instanciée par le filtrage et elle sera considérée comme une variable d'élément pour les besoins du filtrage ;
- si $(\mathtt{x},??) \in C$ alors `x` n'est pas une variable définie dans le contexte mais sera instanciée par le filtrage et elle sera considérée comme une variable de liste pour les besoins du filtrage.

Les trois autres valeurs que retourne $Conv$ sont :

- Un terme sur lequel la notion de filtrage peut être appliquée.
- Une expression booléenne qui correspond à la traduction en JAVA des filtrages sur le type des motifs ; cette expression booléenne doit s'évaluer en `true` pour que la solution du filtrage soit acceptée.
- Un bloc d'instructions qui correspond à la traduction en JAVA des étiquettages ; lorsqu'un filtre est valide pour le motif, ce bloc d'instructions doit être exécuté avant l'action afin que les étiquettes soient déclarés et instanciées.

Définissons $Conv$:

$Conv$:

$\mathsf{C}^+ \times P \mapsto \mathsf{C}^+ \times T \times B \times I$

C, `[| |]` $\rightarrow C, <>, \texttt{true},$;

C, `[|` x_1 `, ... ,` x_n `|]` $\rightarrow C', < S_1, \ldots, S_n >, b, i$ tel que $C_1, S_1, b_1, i_1 = Conv(C, x_1), \ldots, C_n, S_n, b_n, i_n = Conv(C, x_n)$ et $C' = C_1 \cup_c \ldots \cup_c C_n$ et $b = b_1$ `&&` $\ldots$ `&&` b_n et $i = i_1 \; \ldots i_n$

C, `?x` $\rightarrow C', \texttt{x}, \texttt{true},$; tel que $C' = \{(y, e) | y \neq \texttt{x}, (y, e) \in C\} \cup_c \{(\texttt{x}, ?)\}$

C, `??x` $\rightarrow C', \texttt{x}, \texttt{true},$; tel que $C' = \{(y, e) | y \neq \texttt{x}, (y, e) \in C\} \cup_c \{(\texttt{x}, ??)\}$

C, `_` $\rightarrow C \cup_c \{(x, ?)\}, x, \texttt{true},$; où x est une nouvelle variable

C, `...` $\rightarrow C \cup_c \{(x, ??)\}, x, \texttt{true},$; où x est une nouvelle variable

C, `_`(l) $\rightarrow C \cup_c \{(x, ?)\}, x, \texttt{true}, i$ où x est une nouvelle variable et tel que $i = x_1$`=` x`.`m_1 ; $\ldots x_n$`=` x`.`m_n ; si $l = x_1$ `:`m_1 `, ... ,` x_n `:`m_n

C, `?x`(l) $\rightarrow C', \texttt{x}, \texttt{true}, i$ tel que $C' = \{(y, e) | y \neq \texttt{x}, (y, e) \in C\} \cup_c \{(\texttt{x}, ?)\}$ et tel que $i = x_1$`=` x`.`m_1 ; $\ldots x_n$`=` x`.`m_n ; si $l = x_1$ `:`m_1 `, ... ,` x_n `:`m_n

C, `t`(l) $\rightarrow C, x, b, i$ où x est une nouvelle variable et tel que $b = x$ `instanceof` t et tel que $i = x_1$`=` x`.`m_1 ; $\ldots x_n$`=` x`.`m_n ; si $l = x_1$ `:`m_1 `, ... ,` x_n `:`m_n

C, `?x:t`(l) $\rightarrow C', \texttt{x}, b, i$ tel que $C' = \{(y, e) | y \neq \texttt{x}, (y, e) \in C\} \cup_c \{(\texttt{x}, ?)\}$ et tel que $b =$ `x instanceof` t et tel que $i = x_1$`=` x`.`m_1 ; $\ldots x_n$`=` x`.`m_n ; si $l = x_1$ `:`m_1 `, ... ,` x_n `:`m_n

$C, x \rightarrow C, x, \texttt{true},$; si aucune autre règle ne s'applique

Nous utiliserons $\rightarrow^*_{JMod}$ qui représente la relation d'évaluation des expressions booléennes JavaModulo. Cette évaluation peut provoquer des effets de bord ou lever des exceptions ; dans notre sémantique nous supposons qu'un tel cas de figure ne se produit pas. Cette relation dépend du contexte : une variable s'évalue en sa valeur dans le contexte.

Les motifs peuvent être conditionnels ou non ; nous ne donnerons pas de sémantique pour les motifs non-conditionnels puisqu'il suffit de les considérer comme conditionnels avec la condition `true`. Donnons maintenant une sémantique opérationnelle de l'évaluation des constructions `match` et `matchall`. Les règles d'inférence données s'appliquent à des séquents : en partie gauche du séquent se trouve le contexte, en partie droite l'ensemble des instructions restantes à exécuter. On peut munir de la sorte n'importe quel ensemble d'instructions par un contexte, de sorte qu'on puisse donner un contexte dont la portée est limitée.

1. Lorsque les constructions `match` et `matchall` n'ont plus de motifs à filtrer, l'exécution passe aux instructions suivantes :
$$\frac{C \vdash \texttt{match(}s\texttt{)\{\}} \; \textit{instructions}}{C \vdash \textit{instructions}}$$
$$\frac{C \vdash \texttt{matchall(}s\texttt{)\{\}} \; \textit{instructions}}{C \vdash \textit{instructions}}$$
2. Lorsqu'un motif filtre le sujet dans une construction `match`, les étiquetages puis l'action sont exécutés avec un nouveau contexte instancié par le filtre ; l'exécution reprendra ensuite sur les instructions suivantes avec le contexte original :
$$\frac{C \vdash \texttt{match(}s\texttt{)\{}\textit{motif} \; \texttt{if} \; \textit{cond} \; \texttt{->} \; \textit{action} \; \textit{suite}\texttt{\}} \; \textit{instr}}{(C' \vdash i \;\; \textit{action}) \;\; (C \vdash \textit{instr})}$$
avec $C^+, \textit{motif}', b, i = Conv(C, \textit{motif})$, si $\textit{motif}'$ filtre $*(s)$ dans C^+ avec le filtre σ, que C' instancie C^+ par σ, si $C' \vdash \textit{cond} \rightarrow^*_{JMod} \texttt{true}$ et si $C' \vdash b \rightarrow^*_{JMod} \texttt{true}$
3. Lorsqu'un motif filtre n fois le sujet dans une construction `matchall`, les étiquetages puis l'action sont exécutés n fois avec à chaque fois un nouveau contexte instancié par le filtre correspondant ; l'exécution reprendra ensuite sur le filtrage des motifs suivants avec le contexte original :
$$\frac{C \vdash \texttt{matchall(}s\texttt{)\{}\textit{motif} \; \texttt{if} \; \textit{cond} \; \texttt{->} \; \textit{action} \; \textit{suite}\texttt{\}} \; \textit{instr}}{(C_1 \vdash i \;\; \textit{action}) \ldots (C_n \vdash i \;\; \textit{action}) \;\; (C \vdash \; \texttt{matchall(}s\texttt{)\{}\textit{suite}\texttt{\}} \; \textit{instr})}$$

$$\begin{array}{c}\text{avec } C^+, motif', b, i = Conv(C, motif),\\ \text{si } motif' \text{ filtre } *(s) \text{ dans } C^+ \text{ avec } n \text{ filtres différents } \sigma_1, \ldots, \sigma_n,\\ \text{que } C_1, \ldots, C_n \text{ instancient } C^+ \text{ respectivement par } \sigma_1, \ldots, \sigma_n,\\ \text{si } C_1 \vdash cond \rightarrow^*_{JMod} \texttt{true}, \ldots, C_n \vdash cond \rightarrow^*_{JMod} \texttt{true}\\ \text{et si } C_1 \vdash b \rightarrow^*_{JMod} \texttt{true}, \ldots, C_n \vdash b \rightarrow^*_{JMod} \texttt{true}\end{array}$$

4. Lorsqu'un motif ne filtre pas le sujet dans une construction `match` ou `matchall`, l'exécution continue sur le filtrage des motifs suivants avec le contexte original :

$$\frac{C \vdash \texttt{match(}s\texttt{)\{}motif\ \texttt{if}\ cond\ \texttt{->}\ action\ suite\texttt{\}}\ instr}{C \vdash \texttt{match(}s\texttt{)\{}suite\texttt{\}}\ instr}$$

$$\text{avec } C^+, motif', b, i = Conv(C, motif), \text{ si } motif' \text{ ne filtre pas } *(s) \text{ dans } C^+,$$
$$\text{ou} \begin{cases} \text{si } motif' \text{ filtre } *(s) \text{ dans } C^+ \text{ avec le filtre } \sigma,\\ \text{que } C' \text{ instancie } C^+ \text{ par } \sigma\\ \text{et que } (C' \vdash cond \rightarrow^*_{JMod} \texttt{false} \text{ ou } C' \vdash b \rightarrow^*_{JMod} \texttt{false})\end{cases}$$

$$\frac{C \vdash \texttt{matchall(}s\texttt{)\{}motif\ \texttt{if}\ cond\ \texttt{->}\ action\ suite\texttt{\}}\ instr}{C \vdash \texttt{matchall(}s\texttt{)\{}suite\texttt{\}}\ instr}$$

$$\text{avec } C^+, motif', b, i = Conv(C, motif), \text{ si } motif' \text{ ne filtre pas } *(s) \text{ dans } C^+,$$
$$\text{ou} \begin{cases} \text{si } motif' \text{ filtre } *(s) \text{ dans } C^+ \text{ avec } n \text{ filtres différents } \sigma_1, \ldots, \sigma_n,\\ \text{que } C_1, \ldots, C_n \text{ instancient } C^+ \text{ respectivement par } \sigma_1, \ldots, \sigma_n\\ \text{et que} \begin{cases} C_1 \vdash cond \rightarrow^*_{JMod} \texttt{false} \text{ ou } C_1 \vdash b \rightarrow^*_{JMod} \texttt{false} \text{ ou}\\ \ldots\\ C_n \vdash cond \rightarrow^*_{JMod} \texttt{false} \text{ ou } C_n \vdash b \rightarrow^*_{JMod} \texttt{false}\end{cases}\end{cases}$$

5. Implantation

Le compilateur JAVAMODULO se compose de plusieurs pans :

1. La définition et l'implantation d'un nouveau parseur par différence par rapport à celle du parseur de JAVA (ajout de nouvelles règles et suppression ou modification des règles existantes) ainsi que d'un nouveal anaylyseur lexical (*lexer*) puisque JAVAMODULO introduit de nouveaux mots-clés.
2. L'ajout de nouveaux nœuds d'ASA et la modification de certains nœuds d'ASA existants pour modéliser les ajouts de JAVAMODULO à JAVA : les collections, la nouvelle utilisation des opérateurs `+` et `+=` et les constructions `match` et `matchall`.
3. L'ajout de nouvelles passes et la modification de certaines passes de compilation sont nécessaires pour gérer correctement le filtrage et préparer la traduction ; nous donnerons un aperçu des problèmes rencontrés et des solutions apportées.
4. La génération du code en sortie doit être bien choisie et implantée pour donner du code qui soit correctement accepté et compilé par les compilateurs TOM puis JAVA.

5.1. Le parseur de JAVAMODULO

Les modifications du parseur découlent des grammaires BNF données en 3.2 pour le type collection, en 3.3 pour les collections et en 3.4 pour les constructions `match` et `matchall`.

Types de collection : les type références de JAVA sont étendus par les types de collection de JAVAMODULO.

Valeurs de collection : les valeurs primitives de JAVA sont étendues par les valeurs des Séquences.

Instructions `match` et `matchall`, motifs : les instructions (*statements*) de JAVA sont étendues par deux nouvelles instructions, `match` et `matchall`.

5.2. L'ASA de JavaModulo

Nous ne parlerons que des nœuds d'ASA qui ont été créés et des nœuds de Polyglot qui ont été modifiés. Enfin, nous ne parlerons pas que des nœuds d'ASA mais aussi d'autres composants qui s'y rattachent plus ou moins directement, comme les nœuds de type ou le système de type.

Types de collection : Pour représenter chaque nouveau type, deux nouveaux nœuds sont introduits. De plus le système de type est modifié pour tenir compte de ces nouveaux types possibles. Cette modification est légère mais par contre une passe de compilation est ajoutée pour typer les séquences en partie droite d'une déclaration locale et la passe de typage de Polyglot est modifiée pour typer les autres séquences à partir de leur premier élément.

Valeurs de collection : Nouveaux nœuds d'ASA introduits.

Opérateurs sur les collections : Le comportement des opérateurs `+` et `+=` est modifié pour pouvoir utiliser aussi les opérateurs `+` et `+=` sur les collections.

Constructions `match` et `matchall` : Beaucoup de nœuds d'ASA ont été créés pour représenter les instructions `match` et `matchall` et les motifs qu'elles utilisent.

Constructions TOM : Nous avons aussi besoin d'autres nœuds d'ASA pour les besoins des constructions TOM, pour que le code en sortie soit valide pour TOM.

5.3. Le processus de compilation de JavaModulo

Nous allons donner une idée des passes du compilateur JavaModulo, ce qui comprend les passes fournies par Polyglot et celles que nous avons ajoutées ou modifiées. Mis à part la première passe, qui crée l'ASA, et la dernière passe, qui crée le code en sortie, ces passes sont successivement appliquées à l'ASA.

5.3.1. La philosophie de Polyglot

Le processus de compilation de Polyglot est dit *multi-passes* : il se compose de nombreuses passes de compilation successives. Le compilateur JavaModulo se compose de plus de vingt passes. Dans la pratique, une telle architecture de compilation n'est pas inefficace car Polyglot suit et incite ses utilisateurs à suivre une logique de *construction paresseuse* : on ne construit que les nœuds de l'ASA qui sont modifiés.

De plus, une architecture de compilation multi-passes offre une compréhension plus facile du fonctionnement du compilateur puisque les différentes phases de compilation sont clairement séparées et cette architecture est plus modulaire, qui sont des atouts particulièrement intéressants dans le cadre de Polyglot.

5.3.2. Les passes du compilateur JavaModulo

Nous exposons les problèmes rencontrés durant l'implantation du compilateur JavaModulo et la solution apportée (ou qui devrait être apportée lorsque le problème n'est pas résolu) en termes de modifications du processus de compilation de base fourni par Polyglot.

1. Un problème de nommage de variable : il faut donner un nom de variable aux termes anonymes afin de pouvoir filtrer sur leur type et leur appliquer un étiquetage, ce que nous appellerons annotation, et il faut renommer les variables introduites par filtrage dont le nom est déjà utilisé car le compilateur Java n'accepte pas la redéfinition d'un nom de variable déjà définie. Pour résoudre ce problème, une passe de compilation a été ajoutée qui vérifie si les termes introduits par filtrage nécessitent un renommage ou une annotation. Si nécessaire un nom de variable

« frais » (c'est-à-dire non utilisé dans le contexte) est généré et utilisé pour le renommage ou l'annotation. Dans le cas d'un renommage, le changement de nom de variable est propagé dans l'action.

2. La désambigüisation du code JAVAMODULO ne pouvait pas être faite par les étapes de désambigüisation fournies par POLYGLOT : lors de filtrage et à cause des étiquetages le code JAVAMODULO dans les actions ne pouvait être directement désambigüisé, par manque d'information sur les variables introduites par étiquetage.

   Aussi la désambigüisation est faite progressivement par plusieurs passes au lieu d'une seule, les premières ne touchant pas au contenu des actions. Seule la dernière passe de désambigüisation s'occupe de désambigüiser les actions.

3. La vérification des chemins d'exécution de POLYGLOT est erronée pour du code JAVAMODULO : elle ne tient pas correctement compte des constructions `match` et `matchall`. Ainsi la vérification de POLYGLOT que toute méthode qui doit retourner une valeur en retourne bien une et du bon type ne peut pas être effectuée correctement sur du code JAVAMODULO.

   Dans l'implantation actuelle du compilateur cette vérification n'est pas faite, ce qui veut dire qu'il peut accepter de compiler du code incorrect. Pour résoudre le problème il faut ré-implanter la construction des chemins d'exécution en tenant compte des constructions `match` et `matchall`.

4. POLYGLOT est prévu pour produire du code JAVA en sortie. Dans notre cas nous produisons du code JAVA avec des constructions TOM. TOM a certains besoins que nous devions prendre en compte pour produire du code que TOM acceptera : par exemple il faut donner à TOM des constructions pour qu'il puisse faire le lien entre l'implantation des séquences et leur représentation symbolique.

   Dans ce but plusieurs passes ont été ajoutées pour localiser et changer les portions de code qui doivent être considérées comme du code TOM, afin que le code en sortie soit correct pour TOM, et pour ajouter l'ensemble des constructions nécessaires à TOM.

5.4. L'implantation des collections

Dans ce compilateur les collections sont implantées par la classe `Seq` qui hérite de la classe `java.util.ArrayList`. Les collections sont donc implantées par un terme aplati [14] : c'est une façon de rendre implicite l'associativité *built-in*. En effet Peterson et Stickel [15] ont montré que la réécriture modulo l'associativité, lorsque celle-ci n'est pas rendue implicite, nécessite l'ajout d'*extensions* de règles.

Nous aurions pu utiliser une autre implantation, en particulier pour n'importe quelle classe qui implante l'interface `Collection` les changements seraient mineurs voire nuls.

5.5. Exemples

5.5.1. Un tri

Ce premier exemple implante un tri en JAVAMODULO avec utilisation du filtrage associatif sur les Séquences d'`Integer` : lorsque deux éléments ne sont pas dans le bon ordre, ils sont échangés et on recommence cette opération tant que tous les éléments ne sont pas ordonnés. L'utilisation du filtrage associatif permet d'exprimer très facilement l'algorithme :

```
public class BubbleSort {
    public seq[Integer] sortInteger( seq[Integer] list ) {
        match( list ) {
            [| ??a, ?x(valx:longValue()), ??b, ?y(valy:longValue()), ??c |]
                if  valx > valy  ->
```

```
                    return sortInteger( [| a, y, b, x, c |] ) ;
              _   -> return list ;
        }
    }
}
```

Cet exemple utilise l'étiquetage pour récupérer simplement la valeur encapsulée dans les `Integer` de la Séquence : la condition du filtre s'applique directement sur les valeurs.

Voici le code traduit en JAVA avec des constructions TOM par le compilateur JAVAMODULO ; remarquons la déclaration et l'initialisation des étiquettes, avec désencapsulation des valeurs :

```
public class BubbleSort {
    public Seq sortInteger(Seq list) {
        %match(TomSeq list) {
            seqConc(a*, x, b*, y, c*) -> {{
               long valy = ((Integer) `y).longValue();
               long valx = ((Integer) `x).longValue();
               if (valx > valy) {
                   return this.sortInteger(`seqConc(a*, y, b*, x, c*));
               }
            }}
            _ -> {{ return list; }}
        }
    }

    public BubbleSort() { super(); }


// JavaModulo -- Start of TOM aliases

...

// JavaModulo -- End of TOM aliases

}
```

La partie du code « `TOM aliases` » est omise : il s'agit des constructions pour que TOM puisse faire le lien entre une représentation symbolique et l'implantation des Séquences.

5.5.2. Un filtre avec filtrage sur le type

Ce deuxième exemple montre comment on peut utiliser le filtrage sur le type. `Customer` est une classe générique pour représenter des clients ; d'autres classes sous-classent `Customer` pour représenter des clients avec une nationalité spécifique. Ainsi la classe `FrenchCustomer` représente les clients français. Nous avons une Séquence de `Customer` que nous voulons filtrer pour obtenir le lieu de naissance de tous les clients français qui sont plus vieux qu'un certain âge.

```
public class Filter {
    public seq[String] birthLocations( int age,
                                       seq[Customer] customers ) {
        seq[String] locations = [| |] ;

        matchall( customers ) {
            [| ..., FrenchCustomer(custAge:age(), x:birthLocation()), ... |]
                if  custAge > age  -> locations += x ;
```

```
        }

        return locations ;
    }
}
```

Cet exemple utilise les possibilités de JAVAMODULO :
- le filtrage associatif avec `matchall` pour récupérer toutes les solutions du motif ;
- le filtrage sur le type afin de ne filtrer que sur les éléments de type `FrenchCustomer` ;
- l'étiquetage afin de pouvoir poser facilement une condition sur l'âge `age()` du `FrenchCustomer` et récupérer simplement le lieu de naissance `birthLocation()` ;
- manipuler les Séquences via l'opérateur + pour concaténer des Séquences ou l'opérateur += pour ajouter un élément à une Séquence.

Voici le code traduit en JAVA avec des constructions TOM par le compilateur JAVAMODULO ; notons la condition `instanceof` pour filtrer sur le type :

```
public class Filter {
    public Seq birthLocations(int age, Seq customers) {
        Seq locations = new Seq();
        %match(TomSeq customers) {
            seqConc(_*, jmod_type, _*) -> {{
                if (`jmod_type instanceof FrenchCustomer) {
                    int custAge = ((FrenchCustomer) `jmod_type).age();
                    String x = ((FrenchCustomer) `jmod_type).birthLocation();
                    if (custAge > age) { locations.add(x); }
                }
            }}
        }
        return locations;
    }

    public Filter() { super(); }


// JavaModulo -- Start of TOM aliases

...

// JavaModulo -- End of TOM aliases

}
```

5.5.3. Conclusion sur les exemples

Les exemples permettent de mettre en valeur les apports de JAVAMODULO par rapport à JAVA :
- les structures de données intégrées sont plus faciles à définir et à utiliser qu'avec une API,
- elles sont typées statiquement,
- elles peuvent être filtrées directement sans itération, et
- le filtrage donne un accès direct aux champs et aux méthodes.

En résumé, JAVAMODULO apporte plus de puissance aux développeurs JAVA en leur ajoutant des constructions adaptées à leurs habitudes de programmation, ce qui permet de minimiser le temps d'apprentissage du nouveau langage.

De plus l'intégration complète du filtrage dans JAVAMODULO permet de garantir une certaine cohérence du code au travers du typage.

6. Conclusion

Notre implantation de JAVAMODULO montre comment ajouter une nouvelle fonctionnalité à un langage de manière intégrée et naturelle et par un transformateur source-à-source permettant de rester compatible. L'approche extension de langage a permis de réduire considérablement le travail nécessaire comparé à l'implantation d'un nouveau langage en partant de zéro.

Le choix que nous avons fait de proposer le filtrage modulo comme extension de JAVA montre qu'il est possible de mettre facilement à la portée des non-académiciens des concepts prometteurs issus de la recherche.

Le compilateur de JAVAMODULO est encore en cours de développement. Dans sa version actuelle il ne propose que les séquences, à la fois pour des contraites de temps et parce que TOM ne permet pas encore le filtrage modulo AC, mais les autres types de collection s'implantent exactement de la même façon.

Références

[1] James Gosling, Bill Joy, Guy Steele, and Bracha Bracha. *The Java Language Specification Second Edition.* The Java Series. Addison-Wesley, Boston, Mass., 2000.

[2] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot : An extensible compiler framework for java. In *Proceedings of CC 2003 : 12'th International Conference on Compiler Construction.* Springer-Verlag, April 2003.

[3] Polyglot home page. `http ://www.cs.cornell.edu/Projects/polyglot/`.

[4] Claude Kirchner, Hélène Kirchner, and Marian Vittek. *ELAN, User Manual.* INRIA Lorraine & CRIN, Campus scientifique, 615, rue du Jardin Botanique, BP-101, 54602 Villers-lès-Nancy Cedex, France, November 1995.

[5] ELAN home page. `http ://elan.loria.fr/`.

[6] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. The Maude system. *Lecture Notes in Computer Science*, 1631 :240, 1999.

[7] Maude home page. `http ://maude.cs.uiuc.edu/`.

[8] Jean-Louis Giavitto and Olivier Michel. MGS : a Rule-Based Programming Language for Complex Objects and Collections. *Electronic Notes in Theoretical Computer Science*, volume 59, number 4, 2001.

[9] MGS home page. `http ://mgs.lami.univ-evry.fr/`.

[10] Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. A pattern matching compiler for multiple target languages. *LNCS*, 2622 :61–76, May 2003.

[11] TOM home page. `http ://tom.loria.fr/`.

[12] Jean-Pierre Jouannaud and Hélène Kirchner. Completion of a set of rules modulo a set of equations. *SIAM J. Comput.*, 15(4) :1155–1194, 1986.

[13] Deepak Kapur and Paliath Narendran. Complexity of unification problems with associative-commutative operators. *Journal of Automated Reasoning*, 9(2) :261–288, October 1992.

[14] Claude Marché. Normalized rewriting : an alternative to rewriting modulo a set of equations. *Journal of Symbolic Computation*, 21(3) :253–288, 1996.

[15] Gerald E. Peterson and Mark E. Stickel. Complete sets of reductions for some equational theories. *J. ACM*, 28(2) :233–264, 1981.