[SyntaxeContextuelle]

Syntaxe contextuelle du langage Deca

1 Introduction

Ce document décrit la syntaxe contextuelle du langage Deca, c'est-à-dire la notion de programme Deca « bien typé ». Tout programme Deca bien typé doit être compilé en un programme assembleur dont le comportement à l'exécution respecte la sémantique du programme source décrite dans **[Semantique]**. Réciproquement un programme Deca mal typé doit être rejeté par le compilateur avec un message d'erreur approprié.

La vérification contextuelle d'un programme Deca nécessite trois passages sur le programme. En effet, on peut, dans une déclaration de champ ou de paramètre d'une méthode, faire référence à une classe dont la déclaration apparaît *après* son utilisation. Pour cette raison, la première passe consiste à vérifier uniquement le nom des classes et la hiérarchie de classes. Lors de la deuxième passe, on vérifie les déclarations des champs et la signature des méthodes. D'autre part, les méthodes peuvent être mutuellement récursives. Par conséquent, on vérifie le corps des méthodes au cours d'une troisième passe.

Pour chaque passe, les vérifications à effectuer sont spécifiées formellement par une grammaire attribuée exprimée sur la syntaxe abstraite. Autrement dit, chacune de ces grammaires définit un ensemble d'arbres **program** (l'ensemble des arbres acceptées par cette grammaire attribuée) qui est un sous-ensemble de **PROGRAM**.

Par définition, les programmes Deca « bien typés » sont les programmes dont l'arbre de syntaxe abstraite est accepté successivement par chacune des grammaires attribuées de la syntaxe contextuelle.

2 Domaines d'attributs

Dans cette partie sont définis les domaines d'attributs et les opérations sur les attributs.

2.1 Définition des domaines

On définit d'une part des *domaines de base*, et d'autre part des domaines construits à partir des domaines de base à l'aide des opérations suivantes :

- $D_1 \times D_2$: produit cartésien des domaines D_1 et D_2 ;
- $D_1 \to D_2$: domaine des fonctions partielles de D_1 vers D_2 ; le *domaine de définition* d'une fonction f est noté $\mathrm{dom}(f)$;
- D^* : ensemble des listes d'éléments de D ;
- $D_1 \cup D_2$: union des domaines D_1 et D_2 ;
- $f(D_1, \dots D_n)$ est l'ensemble des termes de la forme $f(d_1, \dots d_n)$, avec $(d_1, \dots d_n) \in D_1 \times \dots D_n$.

Soit Symbol le domaine des identificateurs.
Soit Type le domaine des types du langage Deca. Les types du langage Deca comportent void, boolean, float, int et string. De plus, à chaque classe A du programme correspond un type type_class(A). On a également un type null qui représente le type du littéral `Null`.

$$\text{Type} \triangleq \{\text{void}, \text{boolean}, \text{float}, \text{int}, \text{string}, \text{null}\} \cup \text{type\_class}(\text{Symbol})$$

Visibilité est le type énuméré suivant :

$$\text{Visibility} \triangleq \{\text{protected}, \text{public}\}$$

Les identificateurs sont de différentes *natures*. Dans le domaine TypeNature, on distingue les identificateurs de type ou de classe. Dans le domaine ExpNature, on distingue les identificateurs de champ, de paramètre, de variable et de méthode : c'est-à-dire tous les identificateurs qui peuvent apparaître dans une expression Déca.

$$\begin{aligned}\text{TypeNature} &\triangleq \{\text{type}\} \cup \text{class}(\text{Profil})\\ \text{ExpNature} &\triangleq \{\text{param}, \text{var}\} \cup \text{method}(\text{Signature}) \cup \text{field}(\text{Visibility}, \text{Symbol})\end{aligned}$$

Une nature field(public, *name*) représente un champ *public* de la classe *name* ; field(protected, *name*) représente un champ *protected* de la classe *name*.

Une définition est un couple (nature, type).

$$\begin{aligned}\text{TypeDefinition} &\triangleq \text{TypeNature} \times \text{Type}\\ \text{ExpDefinition} &\triangleq \text{ExpNature} \times \text{Type}\end{aligned}$$

Un environnement est une fonction partielle des identificateurs vers les définitions. On distingue les environnements de types et les environnements d'identificateurs dans les expressions.

$$\begin{aligned}\text{EnvironmentType} &\triangleq \text{Symbol} \to \text{TypeDefinition}\\ \text{EnvironmentExp} &\triangleq \text{Symbol} \to \text{ExpDefinition}\end{aligned}$$

Une signature est une liste (ordonnée) de types.

$$\text{Signature} \triangleq \text{Type}^*$$

Une extension de classe est le nom de sa super-classe, ou 0 pour la classe Object qui n'a pas de super-classe.

$$\text{Extension} \triangleq \text{Symbol} \cup \{0\}$$

Un profil de classe est constitué du nom de sa super-classe et d'un environnement qui contient les définitions des champs et méthodes de la classe.

$$\text{Profil} \triangleq \text{Extension} \times \text{EnvironmentExp}$$

Operator est l'ensemble des opérateurs du langage. Les règles (3.49) à (3.63) donnent la correspondance entre les noms d'opérateurs de la syntaxe abstraite et ceux du domaine Operator (utilisés pour spécifier le typage des opérateurs).

$$\text{Operator} \triangleq \{\text{plus}, \text{minus}, \text{mult}, \text{divide}, \text{mod}, \text{eq}, \text{neq}, \text{lt}, \text{gt}, \text{leq}, \text{geq}, \text{and}, \text{or}, \text{not}\}$$

2.2 Opérations et prédicats sur les domaines d'attributs

Notations

— Une fonction partielle f du domaine $D_1 \to D_2$ peut être notée par son graphe, c'est-à-dire "$\{d_1 \mapsto v_1, \ldots d_n \mapsto v_n\}$", où $\text{dom}(f) = \{d_1, \ldots d_n\}$ et $\forall i \in \{1, \ldots n\},\ f(d_i) = v_i$. La fonction nulle part définie est donc notée "$\{\,\}$".

— Un élément du domaine $D_1 \times D_2$ est noté "(d_1, d_2)", ou, s'il n'y a pas d'ambiguïté, "d_1, d_2", avec $d_1 \in D_1$ et $d_2 \in D_2$.

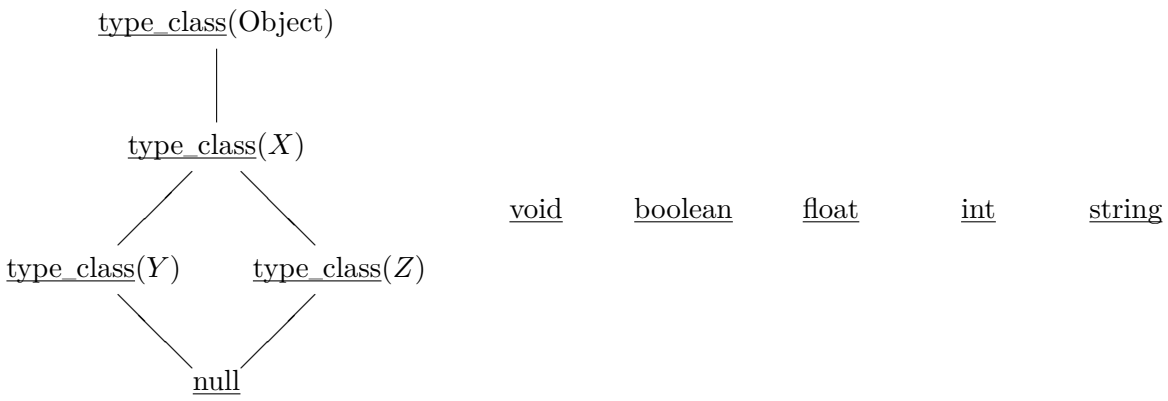
— Un élément du domaine D^* est noté "$[d_1, d_2, \dots d_n]$", où les d_i sont des éléments de D. La séquence vide est notée "[ ]".
On note $e \cdot l$ l'ajout en tête de l'élément e dans la liste l.
$e \cdot [e_1, e_2, \dots e_n] = [e, e_1, e_2, \dots e_n]$.
On note $l \mathbin{@} m$ la concaténation des listes l et m.
$[e_1, e_2, \dots e_n] \mathbin{@} [f_1, f_2, \dots f_p] = [e_1, e_2, \dots e_n, f_1, f_2, \dots f_p]$.

Relation de sous-typage

Soit un environnement de types *env* (de EnvironmentType). La relation de sous-typage relative à l'environnement *env* est définie de la façon suivante :

— Pour tout type T, T est un sous-type de T.
— Pour toute classe A, type_class(A) est un sous-type de type_class(Object).
— Si une classe B étend une classe A dans l'environnement *env*, alors type_class(B) est un sous-type de type_class(A).
— Si une classe C étend une classe B dans l'environnement *env* et si type_class(B) est un sous-type de T, alors type_class(C) est un sous-type de T.
— Pour toute classe A, null est un sous-type de type_class(A).

Par exemple, si on a une classe X et deux classes Y et Z qui sont sous-classes de X, la relation de sous-typage a la forme suivante :



Si T_1 est un sous-type de T_2 relativement à l'environnement *env*, on note subtype(env, T_1, T_2).

Compatibilité pour l'affectation

Soit un environnement de types *env* (de EnvironmentType). On peut affecter à un objet de type T_1 une valeur de type T_2 si l'une des conditions suivantes est satisfaite :

— T_1 est le type float et T_2 est le type int ;
— subtype(env, T_2, T_1).

On note assign_compatible(env, T_1, T_2).

Compatibilité pour la conversion

Soit un environnement de types *env* (de EnvironmentType). On peut convertir une valeur de type T_1 en une valeur de type T_2 si T_1 n'est pas le type void et l'une des conditions suivantes est satisfaite :

— assign_compatible(env, T_1, T_2) ;
— assign_compatible(env, T_2, T_1).

On note cast_compatible(env, T_1, T_2).

Signature des opérateurs

On définit trois opérations partielles : type_unary_op, type_arith_op et type_binary_op, qui permettent de calculer respectivement le type du résultat d'un opérateur unaire, d'un opérateur arithmétique binaire et d'un opérateur binaire quelconque.

type_unary_op : Operator × Type → Type

$$\begin{aligned}
&\text{type_unary_op}(\underline{\text{minus}}, \underline{\text{int}}) &&\triangleq \underline{\text{int}}\\
&\text{type_unary_op}(\underline{\text{minus}}, \underline{\text{float}}) &&\triangleq \underline{\text{float}}\\
&\text{type_unary_op}(\underline{\text{not}}, \underline{\text{boolean}}) &&\triangleq \underline{\text{boolean}}
\end{aligned}$$

type_arith_op : Type × Type → Type

$$\begin{aligned}
&\text{type_arith_op}(\underline{\text{int}}, \underline{\text{int}}) &&\triangleq \underline{\text{int}}\\
&\text{type_arith_op}(\underline{\text{int}}, \underline{\text{float}}) &&\triangleq \underline{\text{float}}\\
&\text{type_arith_op}(\underline{\text{float}}, \underline{\text{int}}) &&\triangleq \underline{\text{float}}\\
&\text{type_arith_op}(\underline{\text{float}}, \underline{\text{float}}) &&\triangleq \underline{\text{float}}
\end{aligned}$$

type_binary_op : Operator × Type × Type → Type

type_binary_op$(op, T_1, T_2) \triangleq$ type_arith_op(T_1, T_2),
si $op \in \{\underline{\text{plus}}, \underline{\text{minus}}, \underline{\text{mult}}, \underline{\text{divide}}\}$

type_binary_op$(\underline{\text{mod}}, \underline{\text{int}}, \underline{\text{int}}) \triangleq \underline{\text{int}}$

type_binary_op$(op, T_1, T_2) \triangleq \underline{\text{boolean}}$,
si $op \in \{\underline{\text{eq}}, \underline{\text{neq}}, \underline{\text{lt}}, \underline{\text{gt}}, \underline{\text{leq}}, \underline{\text{geq}}\}$
et $(T_1, T_2) \in$ dom(type_arith_op)

type_binary_op$(op, T_1, T_2) \triangleq \underline{\text{boolean}}$,
si $op \in \{\underline{\text{eq}}, \underline{\text{neq}}\}$
et $(T_1 = \underline{\text{type\_class}}(A)$ ou $T_1 = \underline{\text{null}})$
et $(T_2 = \underline{\text{type\_class}}(B)$ ou $T_2 = \underline{\text{null}})$

On autorise donc la comparaison de deux objets de classes A et B *quelconques*.

type_binary_op$(op, \underline{\text{boolean}}, \underline{\text{boolean}}) \triangleq \underline{\text{boolean}}$,
si $op \in \{\underline{\text{and}}, \underline{\text{or}}, \underline{\text{eq}}, \underline{\text{neq}}\}$

Le prédicat type_instanceof_op indique si on peut appliquer l'opération `InstanceOf`.

type_instanceof_op : Type × Type → Type
type_instanceof_op$(T_1, T_2) \triangleq \underline{\text{boolean}}$,
si $(T_1 = \underline{\text{type\_class}}(A)$ ou $T_1 = \underline{\text{null}})$
et $T_2 = \underline{\text{type\_class}}(B)$

2.3 Environnements

Définition des environnements prédéfinis

Les identificateurs prédéfinis du langage Deca sont définis par deux environnements : env_types_predef dans EnvironmentType qui définit les types prédéfinis, et env_exp_object de EnvironmentExp qui est associé à la classe prédéfinie Object.

$$\begin{array}{lcll} \text{env\_exp\_object} & \triangleq & \{ & \text{equals} \mapsto (\underline{\text{method}}([\underline{\text{type\_class}}(\text{Object})]), \underline{\text{boolean}}) \quad \} \\ \text{env\_types\_predef} & \triangleq & \{ & \text{void} \mapsto (\underline{\text{type}}, \underline{\text{void}}), \\ & & & \text{boolean} \mapsto (\underline{\text{type}}, \underline{\text{boolean}}), \\ & & & \text{float} \mapsto (\underline{\text{type}}, \underline{\text{float}}), \\ & & & \text{int} \mapsto (\underline{\text{type}}, \underline{\text{int}}), \\ & & & \text{Object} \mapsto (\underline{\text{class}}(0, \text{env\_exp\_object}), \underline{\text{type\_class}}(\text{Object})) \quad \} \end{array}$$

Opérations sur les environnements

On définit deux opérations sur les environnements : l'empilement de deux environnements, noté /, et l'union disjointe de deux environnements, notée $\oplus$. L'opération $\oplus$ est partielle sur les environnements. Dans les définitions ci-dessous, le domaine Environment est soit EnvironmentType, soit EnvironmentExp.

— Empilement

$/ : \text{Environment} \times \text{Environment} \to \text{Environment}$

$$\begin{array}{lcll} \forall x \in \text{Symbol}, \ (env_1/env_2)(x) & \triangleq & env_1(x), & \text{si } x \in \text{dom}(env_1), \\ & \triangleq & env_2(x), & \text{si } x \notin \text{dom}(env_1) \text{ et } x \in \text{dom}(env_2). \end{array}$$

— Union disjointe

$\oplus : \text{Environment} \times \text{Environment} \to \text{Environment}$

$env_1 \oplus env_2$ n'est pas défini si $\text{dom}(env_1) \cap \text{dom}(env_2) \neq \varnothing$.

$$\begin{array}{lcll} \text{Sinon}, \quad \forall x \in \text{Symbol}, \ (env_1 \oplus env_2)(x) & \triangleq & env_1(x), & \text{si } x \in \text{dom}(env_1) \\ & \triangleq & env_2(x), & \text{si } x \in \text{dom}(env_2). \end{array}$$

3 Conventions d'écriture

Les grammaires de chaque passe utilisent le même ensemble de terminaux que celui de **[SyntaxeAbstraite]**. Les non-terminaux de ces grammaires sont soit tout en minuscules, soit tout en majuscules, avec les règles suivantes.

— Un non-terminal en majuscule correspond exactement à un ensemble d'arbres associé au même non-terminal dans la grammaire attribuée de **[SyntaxeAbstraite]**. Les règles associées à la définition de cet ensemble d'arbres ne sont pas répétées.

— Un non-terminal en minuscule qui a exactement son homologue en majuscule dans la grammaire attribuée de **[Decompilation]** définit un sous-ensemble de son homologue.

— Les non-terminaux en minuscule qui n'ont pas d'homologue en majuscule, mais qui ont un homologue en `CamelCase` correspondant à un terminal, représentent des ensembles d'arbres dont la racine est ce nom de nœud.

— Les autres non-terminaux (en minuscule) ne correspondent pas forcément à un ensemble d'arbres : ils peuvent parfois correspondre à des ensembles de "morceaux" d'arbres (en général, un commentaire introduit leur signification).

Par ailleurs, les non-terminaux préfixés par "**list**" désignent toujours des listes d'arbres.

Enfin, on utilise aussi les notations suivantes sur les attributs :

— les attributs synthétisés sont préfixés par $\uparrow$;

— les attributs hérités sont préfixés par $\downarrow$.

3.1 Affectation des attributs

Pour toute règle, les attributs synthétisés du non-terminal en partie gauche et les attributs hérités des non-terminaux en partie droite doivent être affectés. Ces affectations peuvent être effectuées de deux manières différentes : 1. explicitement en utilisant une clause ***affectation*** ; 2. implicitement par une expression fonctionnelle.

1. Affectation explicite de la forme ***affectation*** $v := exp$. Par exemple, la règle (0.1)

   identifier $\downarrow env\_exp \uparrow def$
   $\rightarrow$ `Identifier` $\uparrow name$
   affectation $def := env\_exp(name)$

   signifie qu'à l'attribut synthétisé $\uparrow def$ du non-terminal **identifier** est affecté la valeur $env\_exp(name)$.

2. Affectation implicite par une expression fonctionnelle. L'exemple précédent peut également s'écrire :

   identifier $\downarrow env\_exp \uparrow env\_exp(name)$
   $\rightarrow$ `Identifier` $\uparrow name$.

3.2 Conditions sur les attributs

Les valeurs d'attributs, pour une règle de grammaire, peuvent être contraintes. Ces contraintes peuvent être exprimées de trois manières différentes : 1. explicitement par une condition logique sur les valeurs d'attributs ; 2. lors de l'affectation des attributs, en imposant l'existence d'une valeur (dans le cas des opérations partiellement définies) ; 3. implicitement en contraignant par filtrage les valeurs possibles d'attributs.

1. Utilisation d'une clause ***condition*** P, où P est une condition logique. Si P est faux, la clause n'est pas respectée. Par exemple, la règle (3.28)

   rvalue $\downarrow env\_types \downarrow env\_exp \downarrow class \downarrow type_1$
   $\rightarrow$ **expr** $\downarrow env\_types \downarrow env\_exp \downarrow class \uparrow type_2$
   condition assign_compatible$(env_types, type_1, type_2)$

   impose que les types $type_1$ et $type_2$ sont compatibles pour l'affectation.
   NB : les noms "**rvalue**" et "**lvalue**" viennent de la règle (3.32) sur `Assign`.

2. Par affectation : toute valeur d'attribut doit être définie. Par exemple,

   identifier $\downarrow env\_exp \uparrow def$
   $\rightarrow$ `Identifier` $\uparrow name$
   affectation $def := env\_exp(name)$

   contraint $env\_exp(name)$ à être défini.

3. Par filtrage : on impose une forme particulière pour un attribut hérité dans une partie gauche de règle, ou pour un attribut synthétisé pour une partie droite de règle. Par exemple, la règle (3.29)

   condition $\downarrow env\_types \downarrow env\_exp \downarrow class$
   $\rightarrow$ **expr** $\downarrow env_types \downarrow env_exp \downarrow class \uparrow$boolean

   impose que la valeur de l'attribut synthétisé de **expr** soit le type boolean.
   Dans la règle (3.73)

   rvalue_star $\downarrow env\_types \downarrow env\_exp \downarrow class \downarrow [\,]$
   $\rightarrow$ ε

   on impose que la signature héritée en partie gauche soit la signature vide ($[\,]$).

3.3 Marqueur spécial '_' de noms inutiles

Le symbole '_' est destiné à remplacer un nom de valeur qui n'a qu'une occurrence (et n'est donc pas « utilisé »). Autrement dit ce symbole n'est pas lui-même un nom (chaque occurrence de ce symbole correspond à un nouveau nom inutilisé). Par exemple, la condition de la règle (1.3)

condition $env\_types(super) = (\underline{\text{class}}(\_), \_)$

signifie la même chose que

condition $env\_types(super) = (\underline{\text{class}}(profil), type)$

avec *profil* et *type* deux noms qui n'apparaissent nulle part ailleurs dans le contexte.

3.4 Égalité exprimant une définition

Certaines égalités apparaissant dans les conditions expriment une *définition* de nouveaux noms. C'est typiquement le cas, quand des noms impliqués dans l'égalité ne sont pas contraints ailleurs dans la règle. Par exemple, ces noms apparaîssent uniquement comme attributs hérités dans le membre droit.

Pour faciliter la lecture de ce type d'égalité, on utilise l'opérateur $\triangleq$ qui indique que l'égalité réalise une définition des symboles apparaissant dans son membre gauche. Cet opérateur a bien la même sémantique que $=$. Il sert juste à faciliter la lecture des règles en donnant une indication sur l'effet du $=$.

Par exemple, dans la règle (2.3), la condition utilise une égalité qui réalise une définition de *env_exp_super* :

condition $(\underline{\text{class}}(\_, env\_exp\_super), \_) \triangleq env\_types(super)$

3.5 Règles avec syntaxe spéciale pour les non-terminaux préfixés par list

Comme dans la grammaire de **[Decompilation]**, les règles portant sur les ensembles de listes d'arbres ont une syntaxe spécifique étendant légèrement celle décrite dans la section 1.3 de **[Decompilation]**. En effet,

— elles peuvent maintenant comporter plusieurs attributs hérités ou synthétisés ;

— lorsqu'il n'y a pas d'attribut synthétisé, il n'y a pas d'action d'affectation ;

— l'itération de l'affectation des attributs synthétisés peut être implicite. Ainsi, dans la règle ci-dessous (numérotée (1.2)), env_types_r reçoit initialement la valeur de l'attribut hérité *env_types*, puis à chaque itération sa valeur courante est passée comme attribut hérité de **decl_class**, et il reçoit en sortie de l'itération la valeur de l'attribut synthétisé par ce non-terminal.

list_decl_class $\downarrow env\_types \uparrow env\_types_r$
$\rightarrow$ $\{env\_types_r := env\_types\}$
$[$ (**decl_class** $\downarrow env\_types_r \uparrow env\_types_r)^*$ $]$

On aurait aussi pu écrire cette règle en utilisant un nouvel attribut *tmp* :

list_decl_class $\downarrow env\_types \uparrow env\_types_r$
$\rightarrow$ $\{env\_types_r := env\_types\}$
$[$ (**decl_class** $\downarrow env\_types_r \uparrow tmp$ $\{env\_types_r := tmp\})^*$ $]$

Les règles sur les non-terminaux préfixés par **list** n'utilisent pas de conditions explicites (les conditions explicites sont exprimées dans le non-terminal portant sur les éléments). Par contre, les opérations partielles (comme $\oplus$) expriment des conditions implicites (cf. section 3.2).

4 Règles communes aux trois passes de vérifications contextuelles

Deux règles, communes aux trois passes de vérifications contextuelles, sont détaillées ici.

4.1 Identificateurs dans les expressions

identifier $\downarrow env\_exp\ \uparrow def$ (0.1)
$\rightarrow$ Identifier $\uparrow name$
affectation $def := env\_exp(name)$

On doit trouver une définition associée au nom *name* dans l'environnement *env_exp*.

4.2 Identificateurs de types

type $\downarrow env\_types\ \uparrow type$ (0.2)
$\rightarrow$ Identifier $\uparrow name$
condition $(\_, type) \triangleq env\_types(name)$

5 Grammaire attribuée spécifiant la passe 1

Lors de la passe 1, on vérifie le nom des classes et la hiérarchie de classes. On construit un premier environnement, qui contient env_types_predef, ainsi que les noms des différentes classes du programme. Cet environnement contient un profil incomplet pour chaque classe du programme : celui-ci contient la super-classe, mais ne contient pas l'environnement des champs et méthodes de la classe.

program $\uparrow env\_types$ (1.1)
$\rightarrow$ Program[**list_decl_class** $\downarrow$env_types_predef $\uparrow env\_types$ **MAIN**]

list_decl_class $\downarrow env\_types\ \uparrow env\_types_r$ (1.2)
$\rightarrow$ $\{env\_types_r := env\_types\}$
[(**decl_class** $\downarrow env\_types_r\ \uparrow env\_types_r$)*]

decl_class $\downarrow env\_types\ \uparrow \{name \mapsto (\underline{\text{class}}(super, \{\}), \underline{\text{type\_class}}(name))\} \oplus env\_types$ (1.3)
$\rightarrow$ DeclClass[
Identifier $\uparrow name$
Identifier $\uparrow super$
LIST_DECL_FIELD
LIST_DECL_METHOD]
]
condition $env\_types(super) = (\underline{\text{class}}(\_), \_)$

L'identificateur *super* doit être un identificateur de classe préalablement déclaré. L'environnement associé à la classe est laissé vide ({}), et sera complété pendant la passe suivante.

6 Grammaire attribuée spécifiant la passe 2

Lors de la passe 2, on vérifie les champs et la signature des méthodes des différentes classes. On hérite de l'environnement construit au cours de la passe 1 (attribut env_types_0 hérité par le non-terminal **program**). On construit un environnement qui contient les types prédéfinis et les classes du programme : le profil de chaque classe contient le nom de sa super-classe et l'environnement des champs et méthodes de la classe.

6.1 Programmes

program $\downarrow env\_types_0$ $\uparrow env\_types$ (2.1)
$\rightarrow$ Program[**list_decl_class** $\downarrow env\_types_0$ $\uparrow env\_types$ **MAIN**]

list_decl_class $\downarrow env\_types$ $\uparrow env\_types_r$ (2.2)
$\rightarrow$ $\{env\_types_r := env\_types\}$
[(**decl_class** $\downarrow env\_types_r$ $\uparrow env\_types_r$)*]

6.2 Déclarations de classes

Le corps d'une classe (non-terminaux **list_decl_field** et **list_decl_method**) est analysé dans l'environnement *env_types*, qui est l'environnement créé par la passe 1, complété au fur et à mesure avec les définitions complètes de classes. Le non-terminal **list_decl_method** hérite du nom de la super-classe, qui servira à vérifier la signature d'une méthode redéfinie, règle (2.7). Le non-terminal **list_decl_field** hérite en plus du nom de la classe lui-même (*class*), qui servira à construire la définition des champs, règle (2.5). Leurs attributs synthétisés env_exp_m et env_exp_f contiennent respectivement l'environnement des méthodes et celui des champs de la classe.

Pour la règle suivante, si la passe 1 a été correctement effectuée, alors $name \in \text{dom}(env\_types)$. De plus, dans *env_types*, *super* est associé à une définition de classe. La condition sert donc simplement à récupérer l'environnement *env_exp_super* associé à la classe *super*. Ainsi, l'empilement $\{name \mapsto new\_def\}/env\_types$ complète la définition de *name* laissée incomplète par la passe 1. En pratique, une implémentation pourra simplement ajouter les nouvelles définitions à l'environnement contenu dans la définition de classe construite en passe 1. Il n'est pas nécessaire de créer une nouvelle définition de classe et l'empilement d'environnement peut être fait dès la création de la définition de classe en passe 1.

decl_class $\downarrow env\_types$ $\uparrow env\_types_r$ (2.3)
$\rightarrow$ DeclClass[
Identifier $\uparrow name$
Identifier $\uparrow super$
list_decl_field $\downarrow env\_types$ $\downarrow super$ $\downarrow name$ $\uparrow env\_exp_f$
list_decl_method $\downarrow env\_types$ $\downarrow super$ $\uparrow env\_exp_m$
]
condition $(\underline{\text{class}}(\_, env\_exp\_super), \_) \triangleq env\_types(super)$
affectation $new\_def := (\underline{\text{class}}(super, (env\_exp_f \oplus env\_exp_m)/env\_exp\_super), \underline{\text{type\_class}}(name))$
$env\_types_r := \{name \mapsto new\_def\}/env\_types$

6.3 Déclarations de champs

list_decl_field $\downarrow env\_types$ $\downarrow super$ $\downarrow class$ $\uparrow env\_exp_r$ (2.4)
$\rightarrow$ $\{env\_exp_r := \{\}\}$
[(**decl_field** $\downarrow env\_types$ $\downarrow super$ $\downarrow class$ $\uparrow env\_exp$
$\{env\_exp_r := env\_exp_r \oplus env\_exp\}$)*]

Dans la règle suivante, l'attribut *type* contient le type du champ déclaré, L'attribut *visib* vaut protected si le champ est protégé, et vaut public sinon. Par ailleurs, si l'identificateur *name* est déjà défini dans l'environnement des expressions de la super-classe, alors ce doit être un identificateur de champ.

decl_field $\downarrow env\_types\ \downarrow super\ \downarrow class\ \uparrow\{name \mapsto (\underline{\texttt{field}}(visib, class), type)\}$ (2.5)

$\rightarrow$ `DeclField`$\uparrow visib$[
type $\downarrow env\_types\ \uparrow type$
`Identifier` $\uparrow name$
INITIALIZATION
]

condition $type \neq \underline{\texttt{void}}$ `et`,
`si` $\begin{cases} (\underline{\texttt{class}}(\_, env\_exp\_super), \_) \triangleq env\_types(super) \\ \texttt{et}\ env\_exp\_super(name) \text{ est défini} \end{cases}$
`alors` $env\_exp\_super(name) = (\underline{\texttt{field}}(\_, \_), \_)$.

6.4 Déclarations de méthodes

list_decl_method $\downarrow env\_types\ \downarrow super\ \uparrow env\_exp_r$ (2.6)

$\rightarrow$ $\{env\_exp_r := \{\}\}$
[(**decl_method** $\downarrow env\_types\ \downarrow super\ \uparrow env\_exp$
$\{env\_exp_r := env\_exp_r \oplus env\_exp\}$)*]

Si une méthode est redéfinie, alors celle-ci :

— doit avoir la même signature que la méthode héritée ;

— doit avoir pour type de retour un sous-type du type de retour de la méthode héritée.

decl_method $\downarrow env\_types\ \downarrow super\ \uparrow\{name \mapsto (\underline{\texttt{method}}(sig), type)\}$ (2.7)

$\rightarrow$ `DeclMethod` [
type $\downarrow env\_types\ \uparrow type$
`Identifier` $\uparrow name$
list_decl_param $\downarrow env\_types\ \uparrow sig$
METHOD_BODY
]

condition `si` $\begin{cases} (\underline{\texttt{class}}(\_, env\_exp\_super), \_) \triangleq env\_types(super) \\ \texttt{et}\ env\_exp\_super(name) \text{ est défini} \end{cases}$
`alors` $\begin{cases} (\underline{\texttt{method}}(sig_2), type_2) \triangleq env\_exp\_super(name) \\ \texttt{et}\ sig = sig_2 \\ \texttt{et}\ \underline{\texttt{subtype}}(env\_types, type, type_2) \end{cases}$

L'attribut synthétisé du non-terminal **list_decl_param** permet de construire la signature de la méthode à partir du type des paramètres.

list_decl_param $\downarrow env\_types\ \uparrow sig$ (2.8)

$\rightarrow$ $\{sig := [\,]\}$
[(**decl_param** $\downarrow env\_types\ \uparrow type$ $\{sig := sig \mathbin{@} [type]\}$)*]

decl_param $\downarrow env\_types\ \uparrow type$ (2.9)

$\rightarrow$ `DeclParam`[**type** $\downarrow env\_types\ \uparrow type$ `Identifier` $\uparrow\_$]

condition $type \neq \underline{\texttt{void}}$

7 Grammaire attribuée spécifiant la passe 3

Lors de la passe 3, on vérifie les blocs, les instructions, les expressions et les initialisations. On construit un environnement qui contient les champs et les méthodes, ainsi que les paramètres des méthodes et les variables locales.

La plupart des non-terminaux ont un attribut hérité *class* qui représente la classe dans laquelle les déclarations ou les instructions apparaissent. L'attribut *class* a pour valeur 0 lorsque les déclarations ou les instructions apparaissent dans le programme principal. Cet attribut permet :

— de vérifier que `This` n'apparaît pas dans le programme principal et de construire son type, règle (3.43) ;

— de vérifier que, dans une sélection de champ protégé, le type de l'expression est un sous-type de la classe analysée, règle (3.66).

7.1 Programmes

L'environnement hérité *env_types* du non-terminal **program** est l'environnement synthétisé en passe 2.

program ↓*env_types* (3.1)
→ `Program`[**list_decl_class** ↓*env_types* **main** ↓*env_types*]

list_decl_class ↓*env_types* (3.2)
→ [(**decl_class** ↓*env_types*)*]

main ↓*env_types* (3.3)
→ `EmptyMain`

→ `Main` **bloc** ↓*env_types* ↓{} ↓{} ↓*0* ↓void (3.4)

Le non-terminal **bloc** définit un sous-ensemble de "[**LIST_DECL_VAR LIST_INST**]" (c'est un ensemble de paires d'arbres) et permet de factoriser le traitement des nœuds `Main` et `MethodBody`. Dans le cas de `MethodBody`, **bloc** hérite d'un environnement contenant les déclarations de champs et de méthodes de la classe courante, et d'un autre contenant les déclarations de paramètres. Dans le cas de `Main`, ces deux environnements sont vides. L'attribut *class* vaut 0 et l'attribut *return* vaut void car on analyse le programme principal.

7.2 Déclarations de classes

decl_class ↓*env_types* (3.5)
→ `DeclClass` [
`Identifier` ↑*class*
`Identifier` ↑_
list_decl_field ↓*env_types* ↓*env_exp* ↓*class*
list_decl_method ↓*env_types* ↓*env_exp* ↓*class*
]
condition (class(_, *env_exp*), _) ≙ *env_types*(*class*)

Si les deux premières passes ont été effectuées correctement, *env_types* contient forcément une classe de nom *class*. Cette condition sert donc simplement à récupérer l'environnement des champs et méthodes de la classe.

7.3 Déclarations de champs

list_decl_field ↓*env_types* ↓*env_exp* ↓*class* (3.6)
→ [(**decl_field** ↓*env_types* ↓*env_exp* ↓*class*)*]

decl_field ↓*env_types* ↓*env_exp* ↓*class* (3.7)
→ DeclField[
type ↓*env_types* ↑*type*
Identifier ↑_
initialization ↓*env_types* ↓*env_exp* ↓*class* ↓*type*
]

Ci-dessous, le non-terminal **rvalue** correspond au sous-ensemble des expressions compatibles avec le type *type*.

initialization ↓*env_types* ↓*env_exp* ↓*class* ↓*type* (3.8)
→ Initialization [**rvalue** ↓*env_types* ↓*env_exp* ↓*class* ↓*type*]

→ NoInitialization (3.9)

7.4 Déclarations de méthodes

list_decl_method ↓*env_types* ↓*env_exp* ↓*class* (3.10)
→ [(**decl_method** ↓*env_types* ↓*env_exp* ↓*class*)*]

Pour analyser une méthode (règle (3.11)), on analyse d'abord les paramètres : on construit l'environnement des paramètres. Puis, dans le non-terminal **bloc**, on complète cet environnement avec les déclarations locales, avant d'analyser les instructions dans l'environnement résultant de ces déclarations.

decl_method ↓*env_types* ↓*env_exp* ↓*class* (3.11)
→ DeclMethod [
type ↓*env_types* ↑*return*
Identifier ↑_
list_decl_param ↓*env_types* ↑*env_exp_params*
method_body ↓*env_types* ↓*env_exp* ↓*env_exp_params* ↓*class*
↓*return*
]

list_decl_param ↓*env_types* ↑env_exp_r (3.12)
→ **{**$env\_exp_r := \{\}$**}**
[(**decl_param** ↓*env_types* ↑*env_exp*
{$env\_exp_r := env\_exp_r \oplus env\_exp$**}**)*]

decl_param ↓*env_types* ↑$\{name \mapsto (\underline{\text{param}}, type)\}$ (3.13)
→ DeclParam [**type** ↓*env_types* ↑*type* Identifier ↑*name*]

L'attribut *return* contient le type de retour de la méthode. Cet attribut est utilisé par le non-terminal **bloc** pour vérifier les instructions Return, règle (3.24).

method_body ↓*env_types* ↓*env_exp* ↓*env_exp_params* ↓*class* ↓*return* (3.14)
→ MethodBody
bloc ↓*env_types* ↓*env_exp* ↓*env_exp_params* ↓*class* ↓*return*

→ MethodAsmBody [StringLiteral] (3.15)

7.5 Déclarations de variables

Dans les règles suivantes, *env_exp_sup* représente l'environnement de la classe englobante ou l'environnement vide dans le programme principal. L'environnement *env_exp* est l'environnement contenant les variables déclarées (et les paramètres). L'initialisation des variables est analysée dans l'environnement *env_exp*/*env_exp_sup*.

list_decl_var $\downarrow env\_types \downarrow env\_exp\_sup \downarrow env\_exp \downarrow class \uparrow env\_exp_r$ (3.16)
$\rightarrow$ $\{env\_exp_r := env\_exp\}$
[(**decl_var** $\downarrow env\_types \downarrow env\_exp\_sup \downarrow env\_exp_r \downarrow class \uparrow env\_exp_r$)*]

decl_var $\downarrow env\_types \downarrow env\_exp\_sup \downarrow env\_exp \downarrow class \uparrow \{name \mapsto (\underline{\text{var}}, type)\} \oplus env\_exp$ (3.17)
$\rightarrow$ `DeclVar`[
type $\downarrow env\_types \uparrow type$
`Identifier` $\uparrow name$
initialization $\downarrow env\_types \downarrow env\_exp/env\_exp\_sup \downarrow class \downarrow type$
]
condition $type \neq \underline{\text{void}}$

7.6 Blocs et instructions

Les non-terminaux **bloc**, **list_inst** et **inst** ont plusieurs attributs hérités :

- *env_types* qui représente l'environnement des types ;
- *env_exp_sup* représente l'environnement de l'éventuelle classe englobante (pour **bloc** uniquement) ;
- *env_exp* qui représente l'environnement (éventuellement vide) des paramètres dans le cas de **bloc**, ou l'environnement (tout court) pour les autres ;
- *class* qui représente le nom de la classe où apparaît l'instruction, si l'instruction apparaît dans une classe, et 0 sinon ;
- *return* qui représente le type de retour de la méthode, si l'instruction apparaît dans une méthode, et void sinon.

bloc $\downarrow env\_types \downarrow env\_exp\_sup \downarrow env\_exp \downarrow class \downarrow return$ (3.18)
$\rightarrow$ [
list_decl_var $\downarrow env\_types \downarrow env\_exp\_sup \downarrow env\_exp \downarrow class \uparrow env\_exp\_r$
list_inst $\downarrow env\_types \downarrow env\_exp\_r/env\_exp\_sup \downarrow class \downarrow return$
]

list_inst $\downarrow env\_types \downarrow env\_exp \downarrow class \downarrow return$ (3.19)
$\rightarrow$ [(**inst** $\downarrow env\_types \downarrow env\_exp \downarrow class \downarrow return$)*]

Ci-dessous, le non-terminal **print** correspond juste à l'ensemble des 2 noms de nœuds {`Print`,`Println`}. Le non-terminal **list_exp_print** correspond aux listes de paramètres effectifs imprimables.

inst $\downarrow env\_types \downarrow env\_exp \downarrow class \downarrow return$ (3.20)
$\rightarrow$ **expr** $\downarrow env\_types \downarrow env\_exp \downarrow class \uparrow \_$

$\rightarrow$ **print** [**list_exp_print** $\downarrow env\_types \downarrow env\_exp \downarrow class$] (3.21)

→ `IfThenElse`[(3.22)
condition ↓*env_types* ↓*env_exp* ↓*class*
list_inst ↓*env_types* ↓*env_exp* ↓*class* ↓*return*
list_inst ↓*env_types* ↓*env_exp* ↓*class* ↓*return*
]

→ `NoOperation` (3.23)

→ `Return` [**rvalue** ↓*env_types* ↓*env_exp* ↓*class* ↓*return*] (3.24)

condition *return* ≠ void

→ `While` [(3.25)
condition ↓*env_types* ↓*env_exp* ↓*class*
list_inst ↓*env_types* ↓*env_exp* ↓*class* ↓*return*
]

print → `Print` (3.26)

→ `Println` (3.27)

7.7 Expressions

Le non-terminal **rvalue** correspond aux sous-ensembles des expressions compatibles avec le type $type_1$.

rvalue ↓*env_types* ↓*env_exp* ↓*class* ↓$type_1$ (3.28)
→ **expr** ↓*env_types* ↓*env_exp* ↓*class* ↑$type_2$
condition assign_compatible($env\_types, type_1, type_2$)

Le non-terminal **condition** correspond aux sous-ensembles des expressions booléennes.

condition ↓*env_types* ↓*env_exp* ↓*class* (3.29)
→ **expr** ↓*env_types* ↓*env_exp* ↓*class* ↑boolean

Le non-terminal **exp_print** correspond aux ensembles d'expressions imprimables.

list_exp_print ↓*env_types* ↓*env_exp* ↓*class* (3.30)
→ [(**exp_print** ↓*env_types* ↓*env_exp* ↓*class*)*]

exp_print ↓*env_types* ↓*env_exp* ↓*class* (3.31)
→ **expr** ↓*env_types* ↓*env_exp* ↓*class* ↑*type*
condition *type* = int ou *type* = float ou *type* = string

Ci-dessous, le non-terminal **op_bin** (respectivement **op_un**) contient l'ensemble des nœuds correspondant à un opérateur binaire (resp. unaire) de Operator. Ces deux non-terminaux synthétisent l'opérateur correspondant. Le non-terminal **literal** contient l'ensemble des littéraux (plus la constante `Null`). Il retourne le type de ce littéral.

expr ↓*env_types* ↓*env_exp* ↓*class* ↑*type* (3.32)
→ `Assign` [
lvalue ↓*env_types* ↓*env_exp* ↓*class* ↑*type*
rvalue ↓*env_types* ↓*env_exp* ↓*class* ↓*type*
]

→ **op_bin**↑*op* [(3.33)
expr ↓*env_types* ↓*env_exp* ↓*class* ↑$type_1$
expr ↓*env_types* ↓*env_exp* ↓*class* ↑$type_2$
]
affectation $type := \text{type\_binary\_op}(op, type_1, type_2)$

→ **lvalue** ↓*env_types* ↓*env_exp* ↓*class* ↑*type* (3.34)

→ `ReadInt` (3.35)
affectation $type := \text{int}$

→ `ReadFloat` (3.36)
affectation $type := \text{float}$

→ **op_un** ↑*op* [**expr** ↓*env_types* ↓*env_exp* ↓*class* ↑$type_1$] (3.37)
affectation $type := \text{type\_unary\_op}(op, type_1)$

→ **literal** ↑*type* (3.38)

→ `Cast` [(3.39)
type ↓*env_types* ↑*type*
expr ↓*env_types* ↓*env_exp* ↓*class* ↑$type_2$
]
condition $\text{cast\_compatible}(env\_types, type_2, type)$

→ `InstanceOf`[(3.40)
expr ↓*env_types* ↓*env_exp* ↓*class* ↑$type_1$
type ↓*env_types* ↑$type_2$
]
affectation $type := \text{type\_instanceof\_op}(type_1, type_2)$

→ **method_call** ↓*env_types* ↓*env_exp* ↓*class* ↑*type* (3.41)

→ `New` [**type** ↓*env_types* ↑*type*] (3.42)
condition $type = \text{type\_class}(\_)$

→ `This` (3.43)
condition $class \neq 0$
affectation $type := \text{type\_class}(class)$

literal ↑int → `IntLiteral` (3.44)

literal ↑float → `FloatLiteral` (3.45)

literal ↑string → `StringLiteral` (3.46)

literal ↑boolean → `BooleanLiteral` (3.47)

literal ↑null → `Null` (3.48)

op_bin ↑divide → `Divide` (3.49)

op_bin ↑minus → `Minus` (3.50)

op_bin ↑mod → `Modulo` (3.51)

op_bin ↑mult → `Multiply` (3.52)

op_bin ↑plus → `Plus` (3.53)

op_bin ↑and → `And` (3.54)

op_bin ↑or → `Or` (3.55)

op_bin ↑eq → `Equals` (3.56)

op_bin ↑neq → `NotEquals` (3.57)

op_bin ↑gt → `Greater` (3.58)

op_bin ↑geq → `GreaterOrEquals` (3.59)

op_bin ↑lt → `Lower` (3.60)

op_bin ↑leq → `LowerOrEquals` (3.61)

op_un ↑minus → `UnaryMinus` (3.62)

op_un ↑not → `Not` (3.63)

7.8 Expressions affectables (en partie à gauche d'une affectation)

Le non-terminal **field_ident** (resp. **lvalue_ident**) correspond à l'ensemble des noms de champ (resp. identificateurs affectables) dans le contexte. Ces deux non-terminaux synthétisent le type associé à l'identificateur.

lvalue ↓*env_types* ↓*env_exp* ↓*class* ↑*type* (3.64)

→ **lvalue_ident** ↓*env_exp* ↑*type*

→ `Selection` [(3.65)
expr ↓*env_types* ↓*env_exp* ↓*class* ↑type_class($class_2$)
field_ident ↓env_exp_2 ↑public ↑_ ↑*type*
]
condition (class(_, env_exp_2), _) ≙ $env\_types(class_2)$

→ `Selection` [(3.66)
expr ↓*env_types* ↓*env_exp* ↓*class* ↑type_class($class_2$)
field_ident ↓env_exp_2 ↑protected ↑*class_field* ↑*type*
]
condition (class(_, env_exp_2), _) ≙ $env\_types(class_2)$
`et` subtype(env_types, type_class($class_2$), type_class($class$))
`et` subtype(env_types, type_class($class$), type_class($class\_field$))

La règle (3.66) expriment les contraintes relatives à la visibilité des champs protégés. L'intuition et la raison d'être de ces contraintes sont détaillées dans la section 9.

lvalue_ident ↓*env_exp* ↑*type* (3.67)

→ **identifier** ↓*env_exp* ↑(field(_, _), *type*)

→ **identifier** ↓*env_exp* ↑(param, *type*) (3.68)

→ **identifier** ↓*env_exp* ↑(var, *type*) (3.69)

L'attribut synthétisé *visib* de **field_ident** vaut public si le champ est public, et vaut protected sinon. L'attribut synthétisé *class* est le nom de la classe où le champ est déclaré. L'attribut synthétisé *type* représente le type du champ.

field_ident ↓*env_exp* ↑*visib* ↑*class* ↑*type* (3.70)

→ **identifier** ↓*env_exp* ↑(field(*visib*, *class*), *type*)

7.9 Appels de méthode

L'objet sur lequel est invoqué la méthode doit être d'un type correspondant à une classe $class_2$. Le non-terminal **method_ident** correspond à l'ensemble des identificateurs correspondant à un nom de méthode dans l'environnement *env_exp*. Il synthétise la signature *sig* et le type de retour *type* de cette méthode. Le non-terminal **rvalue_star** définit un sous-ensemble de "**EXPR***" correspondant aux suites de paramètres effectifs compatibles avec la signature *sig* retournée par **method_ident**.

method_call $\downarrow env\_types \downarrow env\_exp \downarrow class \uparrow type$ (3.71)
$\rightarrow$ `MethodCall` [
expr $\downarrow env\_types \downarrow env\_exp \downarrow class \uparrow \underline{\text{type\_class}}(class_2)$
method_ident $\downarrow env\_exp_2 \uparrow sig \uparrow type$
[**rvalue_star** $\downarrow env\_types \downarrow env\_exp \downarrow class \downarrow sig$]
]
condition $(\underline{\text{class}}(\_, env\_exp_2), \_) \triangleq env\_types(class_2)$

method_ident $\downarrow env\_exp \uparrow sig \uparrow type$ (3.72)
$\rightarrow$ **identifier** $\downarrow env\_exp \uparrow (\underline{\text{method}}(sig), type)$

rvalue_star $\downarrow env\_types \downarrow env\_exp \downarrow class \downarrow [\,]$ (3.73)
$\rightarrow$ ε

rvalue_star $\downarrow env\_types \downarrow env\_exp \downarrow class \downarrow (type \cdot sig)$ (3.74)
$\rightarrow$ **rvalue** $\downarrow env\_types \downarrow env\_exp \downarrow class \downarrow type$
rvalue_star $\downarrow env\_types \downarrow env\_exp \downarrow class \downarrow sig$

8 Profils d'attributs des symboles non-terminaux et terminaux

8.0 Profils communs aux trois passes

Identificateurs dans les expressions

identifier ↓EnvironmentExp ↑ExpDefinition
`Identifier` ↑Symbol

Identificateurs de types

type ↓EnvironmentType ↑Type

8.1 Passe 1

program ↑EnvironmentType
list_decl_class ↓EnvironmentType ↑EnvironmentType
decl_class ↓EnvironmentType ↑EnvironmentType

8.2 Passe 2

Programmes

program ↓EnvironmentType ↑EnvironmentType
list_decl_class ↓EnvironmentType ↑EnvironmentType

Déclarations de classes

decl_class ↓EnvironmentType ↑EnvironmentType

Déclarations de champs

list_decl_field ↓EnvironmentType ↓Symbol ↓Symbol ↑EnvironmentExp
decl_field ↓EnvironmentType ↓Symbol ↓Symbol ↑EnvironmentExp
`DeclField`↑Visibility

Déclarations de Méthodes

list_decl_method ↓EnvironmentType ↓Symbol ↑EnvironmentExp
decl_method ↓EnvironmentType ↓Symbol ↑EnvironmentExp
list_decl_param ↓EnvironmentType ↑Signature
decl_param ↓EnvironmentType ↑Type

8.3 Passe 3

Programmes

program ↓EnvironmentType
list_decl_class ↓EnvironmentType
main ↓EnvironmentType

Déclarations de classes

decl_class ↓EnvironmentType

Déclarations de champs

list_decl_field ↓EnvironmentType ↓EnvironmentExp ↓Symbol
decl_field ↓EnvironmentType ↓EnvironmentExp ↓Symbol
initialization ↓EnvironmentType ↓EnvironmentExp ↓Symbol ↓Type

Déclarations de méthodes

list_decl_method ↓EnvironmentType ↓EnvironmentExp ↓Symbol
decl_method ↓EnvironmentType ↓EnvironmentExp ↓Symbol
list_decl_param ↓EnvironmentType ↑EnvironmentExp
decl_param ↓EnvironmentType ↑EnvironmentExp
method_body ↓EnvironmentType ↑EnvironmentExp ↑EnvironmentExp ↓Symbol ↓Type

Déclarations de variables

list_decl_var ↓EnvironmentType ↓EnvironmentExp ↓EnvironmentExp ↓Symbol ↑EnvironmentExp
decl_var ↓EnvironmentType ↓EnvironmentExp ↓EnvironmentExp ↓Symbol ↑EnvironmentExp

Blocs et instructions

bloc ↓EnvironmentType ↓EnvironmentExp ↓EnvironmentExp ↓Symbol ↓Type
list_inst ↓EnvironmentType ↓EnvironmentExp ↓Symbol ↓Type
inst ↓EnvironmentType ↓EnvironmentExp ↓Symbol ↓Type

Expressions

rvalue ↓EnvironmentType ↓EnvironmentExp ↓Symbol ↓Type
condition ↓EnvironmentType ↓EnvironmentExp ↓Symbol
list_exp_print ↓EnvironmentType ↓EnvironmentExp ↓Symbol
exp_print ↓EnvironmentType ↓EnvironmentExp ↓Symbol
expr ↓EnvironmentType ↓EnvironmentExp ↓Symbol ↑Type
literal ↑Type
op_bin ↑Operator
op_un ↑Operator

Expressions affectables

lvalue ↓EnvironmentType ↓EnvironmentExp ↓Symbol ↑Type
lvalue_ident ↓EnvironmentExp ↑Type
field_ident ↓EnvironmentExp ↑Visibility ↑Symbol ↑Type

Appels de méthode

method_call ↓EnvironmentType ↓EnvironmentExp ↓Symbol ↑Type
method_ident ↓EnvironmentExp ↑Signature ↑Type
rvalue_star ↓EnvironmentType ↓EnvironmentExp ↓Symbol ↓Signature

9 Note sur les champs protégés

La règle (3.66), qui porte sur les champs protégés en Deca, est relativement délicate à comprendre. Elle dit la chose suivante :

1. le type de l'expression doit être un sous-type de la classe courante ;
2. le type de la classe courante doit être un sous-type de la classe où le champ protégé est déclaré.

Ces deux règles sont inspirées du point 6.6.2.1 des spécifications Java. La principale différence est que Deca n'a pas de notion de package, et s'inspire des règles de visibilité de deux classes Java situées dans des packages différents.

La condition (2) est la plus simple : il faut "qu'on soit dans une sous-classe". La condition (1) permet en fait de ne pas pouvoir détourner la condition (2).

Prenons l'exemple suivant :

```
class A {
   protected int x;
}

class X {
   void m()
   {
      A a = new A();
      println(a.x) ; // Erreur contextuelle : x est protege
   }
}
```

On utilise une classe `A` dans la classe `X` et on cherche à accéder au champ `x` de `a`. Comme il est protégé, on n'a pas le droit (condition (2)). On pourrait essayer de contourner cette interdiction de la façon suivante.

On déclare une classe `B`, sous classe de `A`,

```
class B extends A {
   int getX(A a) {
      return a.x;
   }
}
```

et on modifie la classe `X` :

```
class X {
   void m() {
      A a = new A();
      B b = new B();
      println(b.getX(a)); // Ok du point de vue de la condition (2)
   }
}
```

Si on pouvait faire cela, n'importe quelle classe pourrait accéder au champ de `A`... En fait, on n'a pas le droit de faire cela, grâce à la condition (1), dans la classe `B`, on a :

```
class B extends A {
   int getX(A a) {
      return a.x; // Erreur contextuelle : le type de 'a' (A) n'est pas
                  // un sous-type de B.
   }
}
```

10 Implémentation de l'environnement

Dans cette partie, on montre sur un exemple comment les environnements *env_types* et *env_exp* peuvent être implémentés.

Un environnement est une liste chaînée de tables d'associations identificateur $\mapsto$ définition.

La figure 1 montre les environnements prédéfinis env_types_predef et env_exp_object.

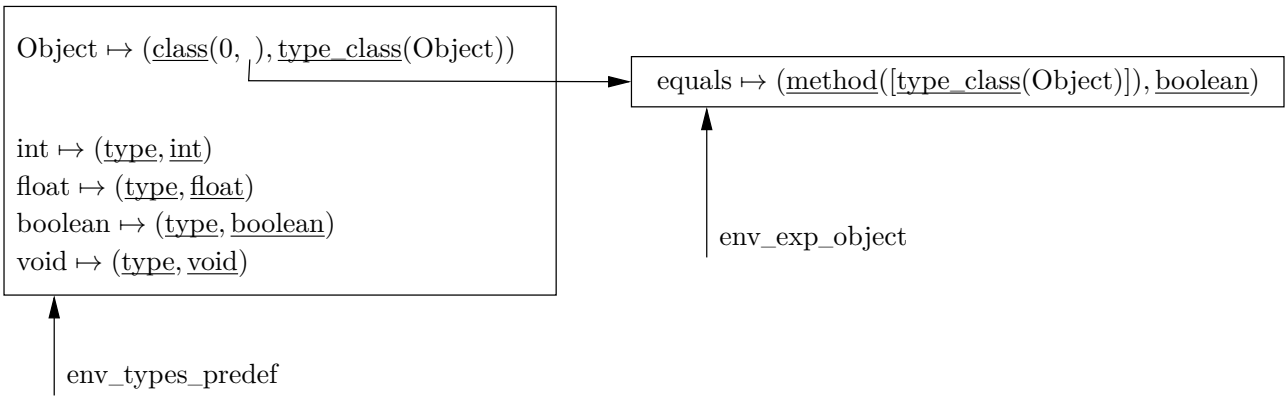


FIGURE 1 – Environnements prédéfinis env_types_predef et env_exp_object.

Considérons le programme Deca suivant.

```
class A {
   protected int x ;
   void setX(int x) {
      this.x = x ;
   }
   int getX() {
      return x ;
   }
   void init() {
      x = 0 ;
   }
}

class B extends A {
   int y ;
   void init() {
      setX(0) ;
      y = 0 ;
  }
}

class C extends A {
   int z ;
   void init() {
      setX(0) ;
      z = 1 ;
   }
}
```

La figure 2 montre l'environnement *env_types* construit à partir des classes du programme, ainsi que l'environnement *env_exp* d'analyse de chaque classe : env_exp_{A}, env_exp_{B} et env_exp_{C}. Cet environnement *env_exp* est l'environnement d'analyse du corps de la classe (attribut hérité de **corps_class**).

La figure 3 montre l'environnement env_exp_{setX} d'analyse de la méthode setX de la classe A. Cette méthode a un paramètre x de type int. Cet environnement correspond à l'attribut hérité *env_exp* de **bloc**.

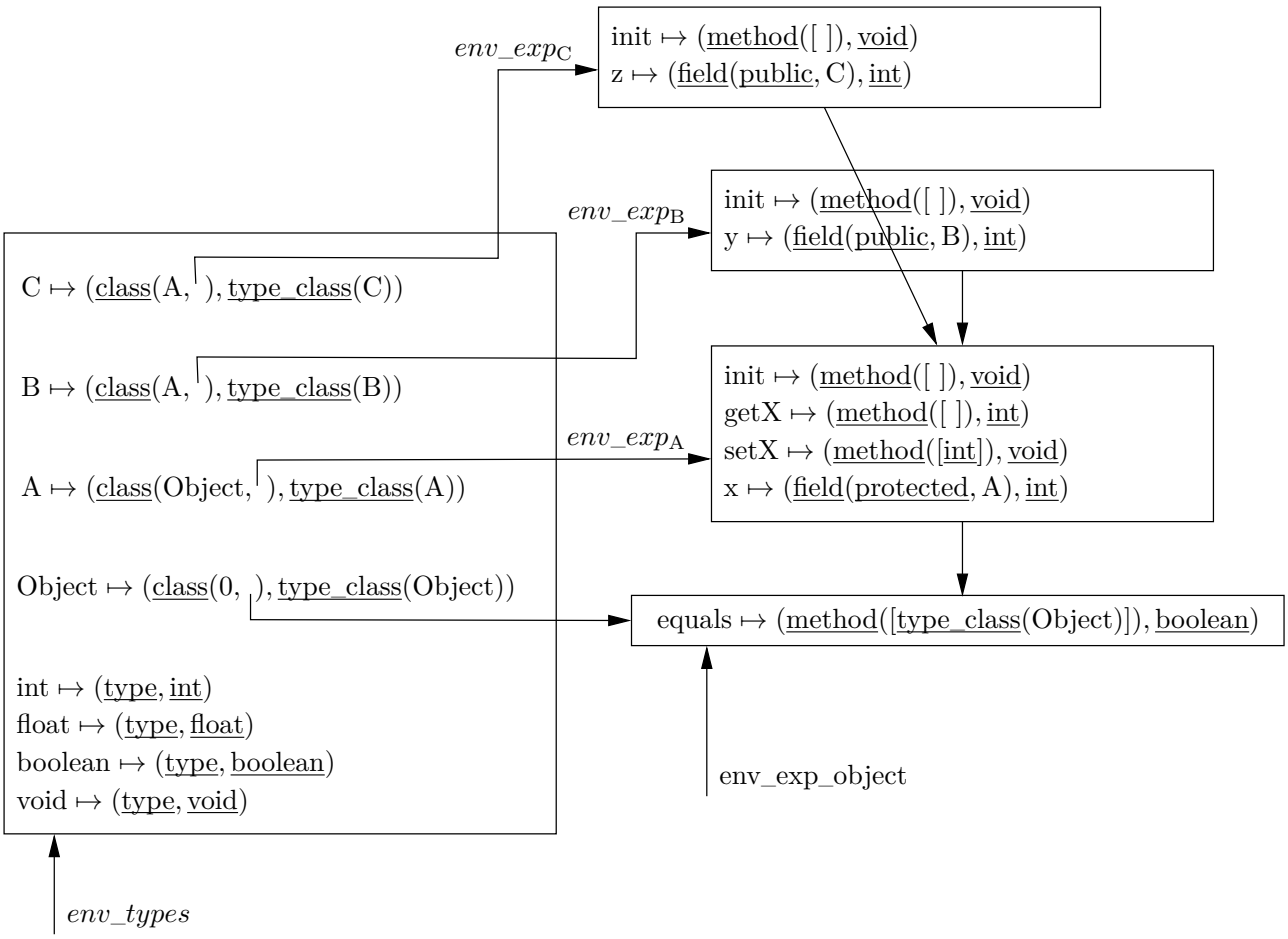


FIGURE 2 – Environnements *env_exp* d'analyse du corps des classes

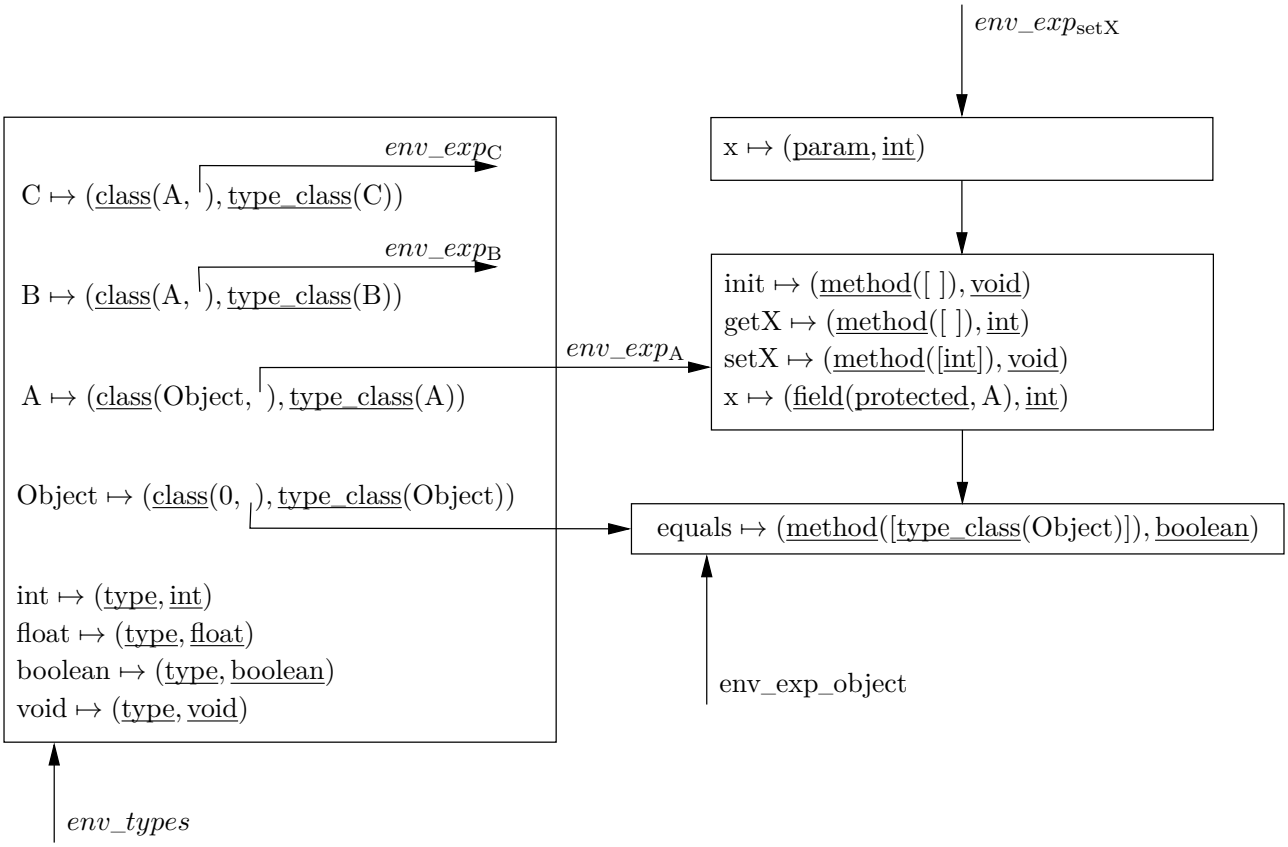


FIGURE 3 – Environnement env_exp_{setX} d'analyse du corps de la méthode setX de A