INGI2339 : Interprétation abstraite.

Projet
Rapport final

Charles-Eric Dessart (SINF22MS)
Jian Hui Lian (SINF22MS)
Cedric Vanderperren (SINF22MS)

Table des matières

Chapitre 1

Sémantique opérationnelle concrète

1.1 Domaines sémantiques

1.1.1 Valeurs ($\mathbb{V}$al)

$$\mathbb{V}\mathrm{al} \triangleq \mathit{Int} + \mathit{Ref} + \{\mathit{null}\} + \{\mathit{noninit}\}$$

Int est l'ensemble des entiers $]-\infty, +\infty[$.
Ref est l'ensemble des références.

1.1.2 Environnement ($\mathbb{E}$nv)

$$\mathbb{E}\mathrm{nv} \triangleq (e : \mathbb{X} + \{\mathit{this}\} \nrightarrow \mathbb{V}\mathrm{al})$$

$\mathbb{X}$ est l'ensemble des variables et paramètres formels.

1.1.3 Store ($\mathbb{S}$tore)

Soit *Inst*, un objet de la forme $\langle n, \langle v_1, ..., v_n\rangle\rangle$, où $n \in \mathbb{N}$ est la taille de l'objet et où $\langle v_1, ..., v_n\rangle \in \mathit{Val}^*$ sont les n valeurs contenues dans l'objet.

Le Store est un ensemble de paires qui peut mathématiquement s'écrire comme suit : $\mathbb{S}\mathrm{tore} \triangleq (s : \mathit{Ref} \nrightarrow \mathit{Inst})$.

1.1.4 Labels ($\mathbb{L}$)

$\mathbb{L}$: Ensembles de labels

Chaque label est associé à une et une seule instruction du programme.

1.1.5 Méthodes ($\mathbb{M}$)

$\mathbb{M}$: Ensembles des identificateurs de méthodes (uniques à chaque niveau de méthode)

1.1.6 Pile ($\mathbb{P}$ile)

$\mathbb{P}\text{ile} \triangleq (l \times e \times x)^*$ où $l \in \mathbb{L}$, $e \in \mathbb{E}\text{nv}$ et $x \in \mathbb{X}$

1.1.7 États

État $= \langle l, e, s, p, in, out \rangle$ où $l \in \mathbb{L}$, $e \in \mathbb{E}nv$, $s \in \mathbb{S}tore$, $p \in \mathbb{P}ile$ et où in et out sont des listes d'entiers.

1.2 Fonctions sémantiques

1.2.1 Conditions ($\mathcal{B}$)

On a $cond ::= sexpr \quad cop \quad sexpr$
Soit la fonction $\mathcal{B} : Cond \rightarrow Env \rightarrow Store \rightarrow \{true, false, error\}$
$\mathcal{B}[\![expr_1 \ cop \ expr_2]\!]_{es} = C[\![cop]\!] v_1 v_2$
$v_1 = V[\![expr_1]\!]_{es}$
$v_2 = V[\![expr_2]\!]_{es}$

Soit la fonction $C : Cop \rightarrow Val \rightarrow Val \rightarrow \{true, false, error\}$

$$C[\![cop]\!] v_1 v_2 = \begin{cases} C_1 v_1 v_2 \text{ si } cop = \text{'<'} \\ C_2 v_1 v_2 \text{ si } cop = \text{'='} \end{cases}$$

Soit la fonction $C_1 : Cop \rightarrow Val \rightarrow Val \rightarrow \{true, false, error\}$

$$C_1 v_1 v_2 = \begin{cases} true \text{ si } v_1 < v_2 \text{ avec } v_1 \text{ et } v_2 \ \in Int \\ false \text{ si } v_1 \geq v_2 \text{ avec } v_1 \text{ et } v_2 \ \in Int \\ error \text{ sinon} \end{cases}$$

Soit la fonction $C_2 : Cop \rightarrow Val \rightarrow Val \rightarrow \{true, false, error\}$

$$C_2 v_1 v_2 = \begin{cases} true \text{ si } v_1 = v_2 \text{ avec } v_1 \text{ et } v_2 \in Int \text{ ou } v_1 \text{ et } v_2 \in Ref \text{ ou } v_1 \text{ et } v_2 = \text{null} \\ false \text{ si } v_1 \neq v_2 \text{ avec } v_1 \text{ et } v_2 \ \in Int \text{ ou } v_1 \text{ et } v_2 \ \in Ref \\ false \text{ si } (v_1 = \{null\} \text{ et } v_2 \neq \{null\} \text{ et } v_2 \in Ref) \text{ ou} \\ \qquad\qquad (v_2 = \{null\} \text{ et } v_1 \neq \{null\} \text{ et } v_1 \in Ref) \\ error \text{ sinon} \end{cases}$$

1.2.2 Désignateurs ($\mathcal{D}$)

On a $des ::= x \mid x.i \mid this.i$
Soit la fonction $\mathcal{D} : Des \rightarrow Env \rightarrow Store \rightarrow \mathbb{X} + (Ref \times \mathbb{N}) + \{error\}$

On obtient :

- $\mathcal{D}[\![x]\!]_{es} = x$

- $\mathcal{D}[\![x.i]\!]_{es} = \begin{cases} \langle r, i\rangle \text{ avec } 1 \leq i \leq n \text{ et } e(x) = r \\ \quad \text{où } r \in \mathit{Ref} \text{ et } s(r) = \langle n, \langle v_1, ..., v_n\rangle\rangle \\ \mathit{error} \text{ si } i < 1 \text{ ou } i > n \text{ ou } e(x) \notin \mathit{Ref} \end{cases}$

- $\mathcal{D}[\![\mathit{this}.i]\!]_{es} = \begin{cases} \langle r, i\rangle \text{ avec } 1 \leq i \leq n \text{ et } e(\mathit{this}) = r \\ \quad \text{où } r \in \mathit{Ref} \text{ et } s(r) = \langle n, \langle v_1, ..., v_n\rangle\rangle \\ \mathit{error} \text{ si } i < 1 \text{ ou } i > n \text{ ou } i > l \\ \quad \text{où } l \text{ est le niveau de la méthode courante.} \end{cases}$

La notation $\langle r, i\rangle$ désigne le $i^{ème}$ champs de l'objet se situant à l'adresse r.

1.2.3 Expressions de droite ($\mathcal{V}$)

On a *expr* ::= *sexpr* | *cexpr*
Soit la fonction $\mathcal{V} : \mathit{Expr} \rightarrow \mathit{Env} \rightarrow \mathit{Store} \rightarrow \mathit{Val} + \{\mathit{error}\}$
On obtient :

- $\mathcal{V}[\![i]\!]_{es} = i$ avec $i \in \mathbb{N}$

- $\mathcal{V}[\![\mathit{this}]\!]_{es} = e(\mathit{this})$

- $\mathcal{V}[\![\mathit{null}]\!]_{es} = \mathit{null}$

- $\mathcal{V}[\![x]\!]_{es} = \mathit{error}$ si $e(x) = \mathit{noninit}$, $e(x)$ sinon

- $\mathcal{V}[\![x.i]\!]_{es} = \begin{cases} v_i \text{ avec } 1 \leq i \leq n \text{ et } e(x) = r \\ \quad \text{où } r \in \mathit{Ref} \text{ et } s(r) = \langle n, \langle v_1, ..., v_n\rangle\rangle \\ \mathit{error} \text{ si } i < 1 \text{ ou } i > n \text{ ou } e(x) \notin \mathit{Ref} \end{cases}$

- $\mathcal{V}[\![\mathit{this}.i]\!]_{es} = \begin{cases} v_i \text{ avec } 1 \leq i \leq n \text{ et } e(\mathit{this}) = r \\ \quad \text{où } r \in \mathit{Ref} \text{ et } s(r) = \langle n, \langle v_1, ..., v_n\rangle\rangle \\ \mathit{error} \text{ si } i < 1 \text{ ou } i > n \text{ ou } e(\mathit{this}) \notin \mathit{Ref} \text{ ou } i > l \\ \quad \text{où } l \text{ est le niveau de la méthode courante.} \end{cases}$

- $\mathcal{V}[\![sexpr1\ aop\ sexpr2]\!]_{es} = \begin{cases} \mathcal{V}[\![sexpr1]\!]_{es}\ +\ \mathcal{V}[\![sexpr2]\!]_{es} \\ \quad \text{si } aop = \text{'+' si } \mathcal{V}[\![sexpr1]\!]_{es} \in \mathit{Int} \text{ et } \mathcal{V}[\![sexpr2]\!]_{es} \in \mathit{Int} \\ \mathcal{V}[\![sexpr1]\!]_{es}\ -\ \mathcal{V}[\![sexpr2]\!]_{es} \\ \quad \text{si } aop = \text{'-' si } \mathcal{V}[\![sexpr1]\!]_{es} \in \mathit{Int} \text{ et } \mathcal{V}[\![sexpr2]\!]_{es} \in \mathit{Int} \\ \mathcal{V}[\![sexpr1]\!]_{es}\ *\ \mathcal{V}[\![sexpr2]\!]_{es} \\ \quad \text{si } aop = \text{'*' si } \mathcal{V}[\![sexpr1]\!]_{es} \in \mathit{Int} \text{ et } \mathcal{V}[\![sexpr2]\!]_{es} \in \mathit{Int} \\ \mathcal{V}[\![sexpr1]\!]_{es}\ /\ \mathcal{V}[\![sexpr2]\!]_{es} \\ \quad \text{si } aop = \text{'/' si } \mathcal{V}[\![sexpr1]\!]_{es} \in \mathit{Int} \text{ et } \mathcal{V}[\![sexpr2]\!]_{es} \in \mathit{Int} \text{ et} \\ \quad V[\![sexpr2]\!]_{es} \neq 0 \\ \mathcal{V}[\![sexpr1]\!]_{es}\ \%\ \mathcal{V}[\![sexpr2]\!]_{es} \\ \quad \text{si } aop = \text{'\%' si } \mathcal{V}[\![sexpr1]\!]_{es} \in \mathit{Int} \text{ et } \mathcal{V}[\![sexpr2]\!]_{es} \in \mathit{Int} \text{ et} \\ \quad V[\![sexpr2]\!]_{es} \neq 0 \\ \mathit{error} \text{ sinon} \end{cases}$

Les opérations arithmétiques définies ci-dessus sont les opérations arithmétiques Java

1.2.4 Affectations ($\mathcal{A}$)

On a $ass ::= \quad des := expr \quad | \quad x := new/i$
Soit la fonction $\mathcal{A} : \mathit{Ass} \to \mathit{Env} \to \mathit{Store} \to \mathit{Env} \times \mathit{Store} + \{\mathit{error}\}$
On obtient :

- $\mathcal{A}[\![x := expr]\!]_{es} = \langle e[x/\mathcal{V}[\![expr]\!]_{es}], s\rangle$ si $\mathcal{V}[\![expr]\!]_{es} \neq \mathit{error}$, error sinon

- $\mathcal{A}[\![x.i := expr]\!]_{es} = \langle e, s[\mathcal{D}[\![x.i]\!]_{es}/\mathcal{V}[\![expr]\!]_{es}]\rangle$ si $\mathcal{D}[\![x.i]\!]_{es} \neq \mathit{error}$ et $\mathcal{V}[\![expr]\!]_{es} \neq \mathit{error}$, error sinon

- $\mathcal{A}[\![this.i := expr]\!]_{es} = \langle e, s[\mathcal{D}[\![this.i]\!]_{es}/\mathcal{V}[\![expr]\!]_{es}]\rangle$ si $\mathcal{D}[\![this.i]\!]_{es} \neq \mathit{error}$ et $\mathcal{V}[\![expr]\!]_{es} \neq \mathit{error}$, error sinon

- Soit la fonction alloc : $\mathbb{N} \to \mathit{Store} \to \mathit{Ref} \times \mathit{Store}$
  $alloc(i, s) = (r, s + (r \to \langle i, \langle v_1, ...v_i\rangle\rangle))$
  avec $\quad \langle v_1, ..., v_i\rangle = \{noninit\}$

  $\mathcal{A}[\![x := new/i]\!]_{es} = \langle e[x/r], s'\rangle$ si $i \geq 0$ avec $(r, s') = alloc(i, s)$

La notation $s[\langle r, i\rangle/v]$ signifie que la valeur du $i^{ème}$ champ de l'objet se situant à l'adresse r est mise à v.

1.2.5 Entrées/Sortie ($\mathcal{I}n, \mathcal{O}ut$)

Soit la fontion $\mathcal{I}n : \mathbb{I}n \rightarrow Env \rightarrow In \rightarrow Env \times In + \{error\}$

$$\mathcal{I}n[\![read\ x]\!]_{es\ in} = \begin{cases} \langle e[x/\mathcal{V}[\![x]\!]], s, in \backslash \{x\} \rangle \text{ si } \mathcal{V}[\![x]\!] \in Int \text{ et } in \text{ non vide} \\ error \text{ sinon} \end{cases}$$

$\mathbb{I}n$: Ensemble des lectures d'un nombre entier
In : Ensemble des listes d'entiers à lire
$in \in In$: une liste d'entiers à lire

Soit la fontion $\mathcal{O}ut : \mathbb{O}ut \rightarrow Env \rightarrow Out \rightarrow Out + \{error\}$

$$\mathcal{O}ut[\![write\ x]\!]_{es\ out} = \begin{cases} \langle e, s, out \cup \mathcal{V}[\![x]\!]_{es} \rangle \text{ si } \mathcal{V}[\![x]\!]_{es} \in Int \\ error \text{ sinon} \end{cases}$$

$\mathbb{O}ut$: Ensemble des écritures d'un nombre entier
Out : Ensemble des listes d'entiers écrits
$out \in Out$: une liste d'entiers écrits

1.3 Relations de transition

1.3.1 Affectations

- $\langle l, e, s, p, in, out \rangle \xrightarrow{l\ x := expr\ l_a} \langle l', e', s, p, in, out \rangle$
  $l' = l_a$
  $e' = \mathcal{A}[\![x := expr]\!]_{es}$

- $\langle l, e, s, p, in, out \rangle \xrightarrow{l\ x.i := expr\ l_a} \langle l', e, s', p, in, out \rangle$
  $l' = l_a$
  $s' = \mathcal{A}[\![x.i := expr]\!]_{es}$

- $\langle l, e, s, p, in, out \rangle \xrightarrow{l\ this.i := expr\ l_a} \langle l', e, s', p, in, out \rangle$
  $l' = l_a$
  $s' = \mathcal{A}[\![this.i := expr]\!]_{es}$

- $\langle l, e, s, p, in, out \rangle \xrightarrow{l\ x := new/i\ l_a} \langle l', e', s', p, in, out \rangle$
  $l' = l_a$
  $(e', s') = \mathcal{A}[\![x := new/i]\!]_{es}$

1.3.2 Entrées/Sorties

- $\langle l, e, s, p, in, out \rangle \xrightarrow{l\ read\ x\ l_a} \langle l', e', s, p, in', out \rangle$
  $l' = l_a$

$(e', in') = \mathcal{I}n[\![read\ x]\!]_{es\ in}$
Si in est vide, on attend qu'un nombre entier soit lu

– $\langle l, e, s, p, in, out\rangle \xrightarrow{l\ write\ x\ l_a} \langle l', e, s, p, in, out'\rangle$
$l' = l_a$
$out' = \mathcal{O}ut[\![write\ x]\!]_{es\ out}$

1.3.3 Instruction *if*

$\langle l, e, s, p, in, out\rangle \xrightarrow{l\ if\ cond\ then\ l_1\ else\ l_2} \langle l', e, s, p, in, out\rangle$

Si $\mathcal{B}[\![cond]\!] = error$, le programme s'arrête.
$l' = \begin{cases} l_1 \text{ si } \mathcal{B}[\![cond]\!] = true \\ l_2 \text{ si } \mathcal{B}[\![cond]\!] = false \end{cases}$

1.3.4 Appels de méthodes statiques

$\langle l, e, s, p, in, out\rangle \xrightarrow{l\ \ x=m_c(y_1,\ldots,y_n)\ \ l_a} \langle l', e', s, p', in, out\rangle$

Méthode choisie : $m_c(z_1, ..., z_n)\ l_0\ stmt\ l_f\ x_f$

$l' = l_0$
$e' = (z_i = \mathcal{V}[\![y_i]\!]_{es}) \quad \forall\, i : 1 \leq i \leq n$
$p' = \langle\langle l_a, e, x\rangle, p\rangle$

1.3.5 Appels de méthodes dynamiques

$\langle l, e, s, p, in, out\rangle \xrightarrow{l\ \ r=x.m_c(y_1,\ldots,y_k)\ \ l_a} \langle l', e', s, p', in, out\rangle$
Si plusieurs méthodes ont le même identificateur m_c , on choisit celle dont le niveau i est le plus proche du type n de l'objet x (avec $0 \leq i \leq n$). Mathématiquement, cela s'exprime comme ceci :
Soit n le type de l'objet x (sa longueur).
$m_c/i(z_1, ..., z_k)\ l_0\ stmt\ l_f\ x_f$ est choisie avec $0 \leq i \leq n$ et $\forall\, j : 0 \leq j \leq n$ tel que $m_c/j(z_1, ..., z_k)$ existe, $j \leq i$

$l' = l_0$
$e' = (z_i = \mathcal{V}[\![y_i]\!]_{es}) \quad \forall\, i : 1 \leq i \leq k$
$e' = e' \oplus (this = V[\![x]\!]_{es})$
$p' = \langle\langle l_a, e, r\rangle, p\rangle$

1.3.6 Appels vers super

$\langle l, e, s, p, in, out\rangle \xrightarrow{l\ \ x=super.m_c(y_1,\ldots,y_n)\ \ l_a} \langle l', e', s, p', in, out\rangle$
On choisit la méthode m_c dont le niveau i est strictement inférieur et le plus proche du niveau j de la méthode courante.
Mathématiquement, cela s'exprime comme ceci :
soit j le niveau de la méthode courante.
$m_c/i(z_1, ..., z_n)\ l_0\ stmt\ l_f\ x_f$ est choisie avec $0 \leq i < j$ et $\forall\, k : 0 \leq k < j$ tel que $m_c/k(z_1, ..., z_n)$ existe, $k \leq i$

$l' = l_0$
$e' = (z_i = \mathcal{V}[\![y_i]\!]_{es}) \quad \forall\, i : 1 \leq i \leq n$
$e' = e' \oplus (this = e(this))$
$p' = \langle\langle l_a, e, x\rangle, p\rangle$

1.3.7 Retour de méthode

$\langle l, e, s, p, in, out\rangle \longrightarrow \langle l', e', s, p', in, out\rangle$ avec
$p = \langle\langle l', e'', x\rangle, p'\rangle$ et $e' = e''[x/\, Val[\![x_f]\!]]$

Chapitre 2

Sémantique opérationnelle abstraite

2.1 Domaines sémantiques

2.1.1 Entiers ($\mathbb{Z}_a$)

$\mathbb{Z}_a \triangleq \{0, +, -\}$

2.1.2 Références ($\mathbb{R}ef_a$)

$\mathbb{R}ef_a \triangleq \mathbb{N}\backslash\{0\}$

2.1.3 Valeurs ($\mathbb{V}al_a$)

$\mathbb{V}al \triangleq Int + Ref + \{null, noninit\}$
$\mathbb{V}al_{base} \triangleq Z_a + Ref_a + \{null, noninit\}$
$\mathbb{V}al_a \triangleq \mathcal{P}(\mathbb{V}al_{base})$

$Cc : \mathbb{V}al_a \rightarrow \mathbb{S}tore \rightarrow \mathcal{P}(\mathbb{V}al)$
$Cc(r_n)_s = (r \mid r \in \mathbb{R}ef, r \in dom(s) \wedge s(r) = n)$

$Cc(0)_s = \{0\}$
$Cc(null)_s = \{null\}$
$Cc(noninit)_s = \{noninit\}$
$Cc(+)_s = \{\mathbb{Z}_0^+\}$
$Cc(-)_s = \{\mathbb{Z}_0^-\}$
$Cc(X)_s = \bigcup_{x \in X}(Cc(x))$

$UB(x_1, x_2) = \{y \in \mathbb{V}al_{base} \mid y \in x_1 \vee y \in x_2\} = x_1 \cup x_2$

2.1.4 Objets ($\mathbb{O}bj_a$)

$\mathbb{O}bj \triangleq \{\langle m, \langle v_1, ..., v_n\rangle\rangle \ \mid m \in \mathbb{N}\backslash\{0\} \wedge \forall\, i \in \{1, ..., n\} : (v_i \in \mathbb{V}al) \wedge m = n\}$
$\mathbb{O}bj_a \triangleq \{\langle m, \langle v_{a1}, ..., v_{an}\rangle\rangle \ \mid m \in \mathbb{N}\backslash\{0\} \wedge \forall\, i \in \{1, ..., n\} : (v_{ai} \in \mathbb{V}al_a) \wedge m = n\}$

$Cc : \mathbb{O}bj_a \rightarrow \mathbb{S}tore \rightarrow \mathbb{O}bj$
$Cc(\langle n, \langle v_{a1}, ..., v_{an}\rangle\rangle_s = \{\langle n, \langle v_1, ..., v_n\rangle \mid v_i \in Cc(v_{ai})_s \,\forall\, i \in \{1, ..., n\}\}$

$UB : Obj_a^2 \nrightarrow Obj_a$
$(\langle n, \langle v_{a1}, ..., v_{an}\rangle, \langle n, \langle v'_{a1}, ..., v'_{an}\rangle\rangle \rightsquigarrow \ \langle n, UB(v_{a1}, v'_{a1}), ..., UB(v_{an}, v'_{an})\rangle$

2.1.5 Environnement ($\mathbb{E}nv_a$)

$\mathbb{E}nv = (e : \mathbb{X} \cup \{this\} \rightarrow \mathbb{V}al)$
$\mathbb{E}nv_a = (e_a : \mathbb{X} \cup \{this\} \rightarrow \mathbb{V}al_a)$

$Cc : Env_a \rightarrow \mathbb{S}tore \rightarrow \mathcal{P}(\mathbb{E}nv)$
$Cc(e_a)_s = \{e : \forall\, x \in dom(e_a), x \rightarrow e(x) \in Cc(e_a(x))_s\}$

2.1.6 Store ($\mathbb{S}tore_a$)

$\mathbb{S}tore \triangleq (s : Ref \nrightarrow Obj)$.
$\mathbb{S}tore_a \triangleq (s_a : Ref_a \nrightarrow Obj_a)$.

$Cc : \mathbb{S}tore_a \rightarrow \mathcal{P}(\mathbb{S}tore)$
$Cc(s_a) = \{s \in \mathbb{S}tore \mid \forall\, r \in dom(s), s_a(r) = \langle n, \langle v_{a1}, ..., v_{an}\rangle\rangle, s(r) = \langle n, \langle v_1, ..., v_n\rangle\rangle \Rightarrow v_i \in Cc(v_{ai})_s \,\forall\, i \in \{1, ..., n\}$

2.1.7 Domaine ($\mathbb{D}_a$)

$\mathbb{D} \triangleq \mathbb{E}\mathrm{nv} \times \mathbb{S}tore$
$\mathbb{D}_a \triangleq \mathbb{E}\mathrm{nv}_a \times \mathbb{S}tore_a$

$Cc : \mathbb{D}_a \rightarrow \mathcal{P}(\mathbb{D})$
$Cc(\langle e_a, s_a\rangle) = \{\langle e, s\rangle \mid s \in Cc(s_a) \wedge e \in Cc(e_a)_s\}$

2.1.8 Pile ($\mathbb{P}ile_a$)

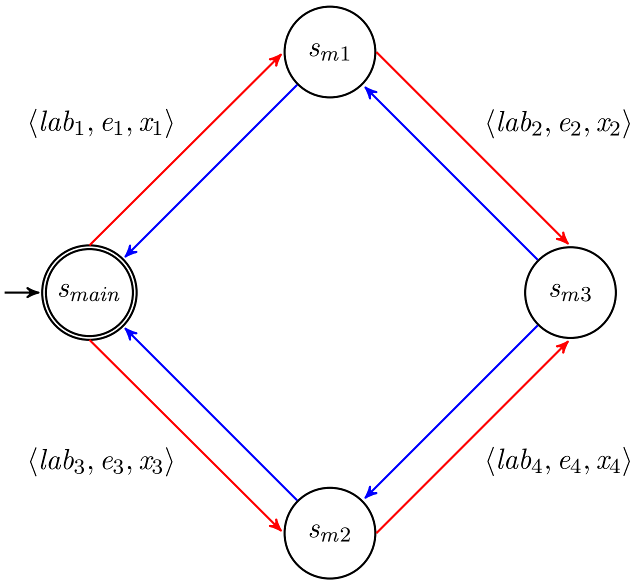
Nous représentons une pile par une machine à états.

$\mathbb{P}ile_a \triangleq \langle S, T, s_o, Ac\rangle$ avec

- S (ensemble des états) où chaque état représente une méthode $\in \mathbb{M}$
- T (ensemble des transitions) où

- chaque transition $\langle s, \langle l, e_a, x\rangle, s'\rangle$ représente un appel de méthode avec
  - s (origine) l'état représentant la méthode appelante
  - $\langle l, e_a, x\rangle$ où $l \in \mathbb{L}$, $e_a \in \mathbb{E}nv_a$ et $x \in \mathbb{X}$
  - s' (cible) l'état réprésentant la méthode appelée
- s_o (état initial) qui est ici l'état représentant la méthode *main* (s_{main})
- Ac (ensemble des états acceptants) qui est ici composé uniquement de l'état représentant la méthode courante (il change donc à chaque appel de méthode)

Illustration



Cette machine a états représente la pile abstraite d'un programme composé de 4 méthodes dont la main. Imaginons qu'il fasse appel à $m1$ qui lui-même fait appel à $m3$. Ensuite, retour dans $m1$, retour dans *main* et appel de $m2$ qui appelle $m3$. Enfin, retour de $m3$ dans $m2$ et aussi dans $m1$ (car on est en abstrait). Enfin, retour dans *main*.

Règle S'il existe une transition $\langle s, lex, s'\rangle$ où $lex \in (\mathbb{L}abel \times \mathbb{E}nv_a \times \mathbb{X})$, alors il n'existe pas de transition $\langle s, lex', s'\rangle$ où $lex \neq lex'$

Remarques

- La machine a états est créée dynamiquement durant l'exécution.
- A l'initialisation, il y a un état pour chaque méthode et aucune transition.
- A chaque appel de méthode, une transition d'appel de méthode vers cet état est ajoutée (s'il n'en existe pas déjà une). Dans le cas où il existe une transition $\langle s, \langle l, e_a, x\rangle, s'\rangle$ et que l'on souhaite ajouter une transition $\langle s, \langle l', e'_a, x'\rangle, s'\rangle$, la transition existante est alors modifiée et devient $\langle s', \langle l', UB(e_a, e'_a), x'\rangle, s\rangle$.
- Lors d'un retour d'une méthode correspondant à un état s à une

méthode correspondant à un état s', l'interpretation abstraite se poursuit parallèlement en effectuant un retour de méthode à chaque méthode correspondant à un état k pour lequel il existe une transition $\langle k, \langle l, env, x\rangle, s\rangle$. L'interprétation se poursuivra donc, entre autres, en effectuant un retour de méthode à la méthode correspondant à s'.

Afin de faire le lien avec la définition d'une pile concrète, nous pouvons dire que
– *Ajouter* $\langle l, env, x\rangle$ *sur la pile* correspond à l'automate qui passe d'un état s à un état s' grâce à une transition $\langle s, \langle l, env, x\rangle, s'\rangle$.
– *Retirer* $\langle l, env, x\rangle$ *sur la pile* correspond à l'automate qui passe d'un état s à un ou des état(s) $s'_1, ..., s'_n (n > 0)$ grâce à une ou des transitions $\langle s'_1, \langle l, env, x\rangle, s\rangle$, ..., $\langle s'_n, \langle l, env, x\rangle, s\rangle$.
– *Tester si la pile est vide* correspond à tester si on a exécuté l'instruction de retour de méthode de la méthode *main*.

Note La variable de retour peut être déduite du label mais l'intégrer dans la définition ci-dessus facilitera la sémantique des retours de méthodes ainsi que l'implémentation.

2.1.9 États ($\mathbb{E}tat_a$)

$\mathbb{E}tat_a = \langle l, e, s, p\rangle$ où $l \in \mathbb{L}$, $e \in \mathbb{E}nv_a$, $s \in \mathbb{S}tore_a$ et $p \in \mathbb{P}ile_a$.

2.2 Fonctions sémantiques

2.2.1 Conditions ($\mathcal{B}$)

On a
– $cond ::= sexpr \quad cop \quad sexpr$
– $\mathcal{B} : Cond \rightarrow D \rightarrow \{true, false, error\}$

Soit la fonction $\mathcal{A}ss : \mathbb{E}xpr \rightarrow \mathbb{E}nv_a \rightarrow \mathbb{S}tore_a \rightarrow \mathcal{P}(\mathbb{Z}_a) \rightarrow \mathbb{E}nv_a \times \mathbb{S}tore_a$
Cette fonction renvoie un domaine abstrait identique au domaine abstrait passé en paramètre à ceci près que la valeur de l'expression est définie par l'ensemble de valeurs abstraites $\mathcal{P}(\mathbb{Z}_a)$.

Spécifications :
– $\mathcal{A}ss[\![x]\!]_{e_a s_a P} = \langle e_{ass}, s_a\rangle$
Où $e_{ass} = e_a \oplus \{x \rightarrow P\}$
– $\mathcal{A}ss[\![i]\!]_{e_a s_a P} = \langle e_a, s_a\rangle$
– $\mathcal{A}ss[\![x.i]\!]_{e_a s_a P} = \langle e_a, s_{ass}\rangle$
Soit : $s_a(e_a(x)) = \{\langle n_i, \langle\langle a_{i1}, .., a_{in}\rangle\rangle\}$

Où : $\forall\, n_i : s_{ass} = s_a \oplus \{n_i \rightarrow \langle n'_i, \langle b_1, .., b_{n'}\rangle\rangle\}$
avec :
- $n'_i = n_i$
- $b_k = a_k \mid \forall\, k = \{1..n\}\backslash i$
- $b_i = P$

– $\mathcal{A}ss[\![this.i]\!]_{e_a s_a P} = \langle e_a, s_{ass}\rangle$
Même chose qu'avec x.i.

Soit la fonction $\mathcal{B}_a : \mathbb{C}ond \rightarrow \mathbb{D}_a \rightarrow \mathcal{P}(\mathbb{D}_a) \times \mathcal{P}(\mathbb{D}_a) \times \{\text{error}, \bot\}$
Cette fonction donne donc un ensemble des domaines abstraits raffinés pour le cas d'une évaluation vraie, un ensemble des domaines abstraits raffinés pour le cas d'une évaluation fausse et un élément indiquant si une erreur peut se réaliser.

Son comportement : $\forall \langle e, s, \rangle \in Cc(\langle e_a, s_a\rangle, \mathcal{B}[\![E_1\, cop E_2]\!]_{es} = v \Rightarrow v \in Cc(\mathcal{B}_a[\![E_1\, cop E_2]\!]_{e_a s_a})$

Spécifications :
$\mathcal{B}_a[\![expr_1 \;=\; expr_2]\!]_{e_a s_a} = \langle T, F, er\rangle$

Soient
- $\mathcal{V}_a[\![expr_1]\!]_{e_a s_a} = \langle L_1, er_1\rangle$
- $\mathcal{V}_a[\![expr_2]\!]_{e_a s_a} = \langle L_2, er_2\rangle$
- $NPZ = \{-0+\}$
- $P \triangleq L_1 \cap L_2$ // Ensemble des valeurs abstraites identiques
- $Q \triangleq NPZ \backslash (P \cap \{0, null\})$

Q est un ensemble de valeurs abstraites que l'on va attribuer comme valeur des termes de la condition dans le cas d'un résultat *false*. Les valeurs contenues dans cet ensemble sont identiques à celles contenues dans, respectivement, L_1 et L_2 à la différence que l'on a enlevé les valeurs 0 et *null* si celles-ci se trouvent à la fois dans dans L_1 et L_2. En effet, les seuls cas où on peut affirmer que le résultat ne sera pas *false* sont quand on compare l'égalité entre 0 et 0 ou entre *null* et *null*.

On a donc :
- $\langle e_t, s_t\rangle \in T$
  Où
  - $\langle e'_t, s'_t\rangle = Ass_{expr_1\, e_a s_a P}$
    On assigne à l'expression de gauche de la condition les valeurs abstraites de P qui sont les valeurs abstraites que l'on retrouve à la fois dans L_1 et L_2.
  - $\langle e_t, s_t\rangle = Ass_{expr_2\, e'_t s'_t P}$
    On assigne à l'expression de droite de la condition les valeurs abstraites de P qui sont les valeurs abstraites que l'on retrouve à la fois

dans L_1 et L_2. On réutilise le domaine abstrait résultat de l'assignation ci-dessus.

- $\langle e_f, s_f\rangle \in F$
  Où
  - $\langle e'_f, s'_f\rangle = Ass_{expr_1\, e_a s_a\, Q}$
  - $\langle e_f, s_f\rangle = Ass_{expr_2\, e'_f s'_f\, Q}$

  Cependant, ces deux heureustiques n'arrivent pas à raffiner le domaine abstrait dans certain cas. Si $P = \varnothing \wedge Q = \varnothing$, alors $\langle e_a, s_a\rangle \in F \wedge \langle e_a, s_a\rangle \in T$

- $er = \begin{cases} \perp \text{ si } L_1 \cup L_2 \subseteq \mathbb{Z}_a \vee L_1 \cup L_2 \subseteq \{null\} \vee L_1 \cup L_2 \subseteq \mathbb{R}ef_a \\ error \text{ sinon} \end{cases}$

Si les valeurs abstraites des termes de la condition font soit toutes partie de $\mathbb{Z}_a$, soit toutes partie de $\mathbb{R}ef_a$ ou soit toutes partie de $\{null\}$, alors il n'y a pas d'erreur.

$\mathcal{B}_a[\![expr_1 \ < \ expr_2]\!]_{e_a s_a} = \langle T, F, er\rangle$

Soient
- $\mathcal{V}_a[\![expr_1]\!]_{e_a s_a} = \langle L'_1, er_1\rangle$
- $\mathcal{V}_a[\![expr_2]\!]_{e_a s_a} = \langle L'_2, er_2\rangle$
- $L_1 \triangleq L'_1 \cap \mathbb{Z}_a$
- $L_2 \triangleq L'_2 \cap \mathbb{Z}_a$

Nous avons à présent filtré les valeurs abstraites résultant des évaluations de expr1 et de expr2 aux valeurs abstrayant un entier. Nous pouvons alors aisément dire dans quel cas une erreur peut se produire :
$L_1 \neq L'_1 \Rightarrow er = error$
$L_2 \neq L'_2 \Rightarrow er = error$

Pour chaque valeur de vérité, la fonction $\mathcal{B}_a$ renvoie exactement 1 ou 2 domaines abstraits raffinés. En effet, dans des cas comme une expr1 égale à $\{-, 0\}$ et une expr2 égale à $\{0, +\}$, il y a deux raffinements de domaines abstraits possibles afin de garder de l'information tout en évitant de n'attribuer qu'une valeur abstraite de $\mathbb{V}al_{base}$ à chaque expression. Dans cet exemple, un domaine aura pour valeur de expr1 $\{-, 0\}$ et expr2 $\{+\}$; un autre aura pour valeur de expr1 $\{-\}$ et expr2 $\{0, +\}$. On peut bien sûr trouver d'autres valeurs pour expr1 et expr2 qui conduiraient aussi à deux raffinements.

Soit la fonction $\mathcal{C}' : \mathbb{Z}_a \rightarrow \mathbb{Z}_a \rightarrow \{T, F\}$ dont la définition est dans le tableau plus bas.

Soient
- $P_{t1} \triangleq \{p \in L_1 \mid \exists\, q \in L_2, \{T\} = \mathcal{C}'_{pq}\}$
- $Q_{t1} \triangleq \{q \in L_2 \mid \nexists p \in L_1, \{F\} = \mathcal{C}'_{pq}\}$
- $P_{t2} \triangleq \{p \in L_1 \mid \nexists q \in L_2, \{F\} = \mathcal{C}'_{pq}\}$
- $Q_{t2} \triangleq \{q \in L_2 \mid \exists\, p \in L_1, \{T\} = \mathcal{C}'_{pq}\}$

P_{t1} et Q_{t1} identifient respectivement un raffinement pour expr1 et un pour expr2 dans le cas d'une évaluation vraie.
P_{t2} et Q_{t2} identifient respectivement un (autre) raffinement pour expr1 et un pour expr2 dans le cas d'une évaluation vraie.

- $P_{f1} \triangleq \{p \in L_1 \mid \exists\, q \in L_2, \{F\} = \mathcal{C}'_{pq}\}$
- $P_{f2} \triangleq \{p \in L_1 \mid \nexists q \in L_2, \{T\} = \mathcal{C}'_{pq}\}$
- $Q_{f1} \triangleq \{q \in L_2 \mid \nexists p \in L_1, \{T\} = \mathcal{C}'_{pq}\}$
- $Q_{f2} \triangleq \{q \in L_2 \mid \exists\, p \in L_1, \{F\} = \mathcal{C}'_{pq}\}$

P_{f1} et Q_{f1} identifient respectivement un raffinement pour expr1 et un pour expr2 dans le cas d'une évaluation fausse.
P_{f2} et Q_{f2} identifient respectivement un (autre) raffinement pour expr1 et un pour expr2 dans le cas d'une évaluation fausse.

Dans certains cas, il n'y a qu'un raffinement possible, ce qui aura pour conséquence d'avoir une de ces valeurs vide. Ces cas-là ne sont pas pris en compte pour renvoyer les domaines abstraits de $\mathcal{B}_a$.

On a :
- $\forall\, P_{ti}, Q_{ti} \mid P_{ti} \neq \varnothing \wedge Q_{ti} \neq \varnothing (i \in \{1, 2\})$

  Soient
  - $\langle e'_{ti}, s'_{ti}\rangle = Ass_{e_a s_a\, expr_i\, P_{ti}}$
  - $\langle e_{ti}, s_{ti}\rangle = Ass_{e'_{ti} s'_{ti}\, expr_i\, Q_{ti}}$

  $\langle e_{ti}, s_{ti}\rangle\ \in T$

- $\forall\, P_{fi}, Q_{fi} \mid P_{fi} \neq \varnothing \wedge Q_{fi} \neq \varnothing (i \in \{1, 2\})$

  Soient
  - $\langle e'_{fi}, s'_{fi}\rangle = Ass_{e_a s_a\, expr_i\, P_{fi}}$
  - $\langle e_{fi}, s_{fi}\rangle = Ass_{e'_{fi} s'_{fi}\, expr_i\, Q_{fi}}$

  $\langle e_{fi}, s_{fi}\rangle\ \in F$

Notons que si L_1 et / ou L_2 ont été filtrés au point qu'ils deviennent des ensembles vides, certains des ensembles P_{xx} et Q_{xx} seront alors vides eux aussi, ce qui aura des conséquences sur ce qui est est écrit ci-dessus. Dans le cas où aucune heureustique n'arrive à raffiner le domaine abstrait, tout comme avec l'égalité on réutilise ce domaine abstrait non raffiné dans les ensembles T et F.

$\mathcal{C}'_{pq}$	`{-}`	`{0}`	`{+}`	v_{a2}
`{-}`	`{T,F}`	`{T}`	`{T}`	
`{0}`	`{F}`	`{F}`	`{T}`	
`{+}`	`{F}`	`{F}`	`{T,F}`	
v_{a1}				

FIGURE 2.1 – $\mathcal{C}'_{pq}$

2.2.2 Désignateurs ($\mathcal{D}_a$)

On a

- $des ::= x \mid x.i \mid this.i$
- $\mathcal{D} : Des \rightarrow Env \rightarrow Store \rightarrow \mathbb{X} + (Ref \times \mathbb{N}) + \{error\}$

Soit la fonction $\mathcal{D}_a : Des \rightarrow \mathbb{E}nv_a \rightarrow \mathbb{S}tore_a \rightarrow (\mathbb{X} + \mathcal{P}(Ref \times \mathbb{N})) \times \{\perp, error\}$

On obtient :

- $\mathcal{D}[\![x]\!]_{e_a s_a} = \langle x, \perp\rangle$
- $\mathcal{D}[\![x.i]\!]_{e_a s_a} = \langle\{\langle r_n, i\rangle \mid n \geq i \wedge r_n \in e_a(x)\}, er\rangle$
  où $er = \begin{cases} \perp \text{ si } \forall\, y \in Ref_a \wedge y = r_n \wedge n \geq i \wedge e_a(x) \subseteq \{\ r_n \mid n \geq i\ \} \\ error \text{ sinon} \end{cases}$
- $\mathcal{D}[\![this.i]\!]_{e_a s_a} = \langle\langle r_n, i\rangle, er\rangle$
  où
  - $1 \leq i \leq n \wedge e_a(this) = r_n \wedge r_n \in Ref_a \wedge s_a(r_n) = \langle n, \langle v_{a_1}, .., v_{a_i}, .., v_{a_n}\rangle\rangle$
  - $er = \begin{cases} \perp \text{ si } i > 0 \wedge i \leq n \wedge e_a(this) \in Ref_a \\ error \text{ sinon} \end{cases}$

La notation $\langle r, i\rangle$ désigne le $i^{ème}$ champs de l'objet se situant à l'adresse r.

2.2.3 Expressions de droite ($\mathcal{V}$)

On a

- $expr ::= sexpr \mid cexpr$
- la fonction $\mathcal{V} : Expr \rightarrow Env \rightarrow Store \rightarrow Val + \{error\}$

Soit la fonction $\mathcal{V}_a : Expr_a \rightarrow Env_a \rightarrow Store_a \rightarrow Val_a \times \{error, \perp\}$
On obtient :

- $\mathcal{V}_a[\![i]\!]_{e_a s_a} = \begin{cases} \langle \{-\}, \perp \rangle \text{si } i < 0 \\ \langle \{0\}, \perp \rangle \text{si } i = 0 \\ \langle \{+\}, \perp \rangle \text{si } i > 0 \end{cases}$
- $\mathcal{V}_a[\![this]\!]_{e_a s_a} = \langle e_a(this) \backslash \{noninit\}, er \rangle \text{où } er = \begin{cases} \perp \text{ si } noninit \notin e_a(x) \\ error \text{ sinon} \end{cases}$
- $\mathcal{V}_a[\![null]\!]_{e_a s_a} = \langle \{null\}, \perp \rangle$
- $\mathcal{V}_a[\![x]\!]_{e_a s_a} = \langle e_a(x) \backslash \{noninit\}, er \rangle \text{où } er = \begin{cases} \perp \text{ si } noninit \notin e_a(x) \\ error \text{ sinon} \end{cases}$
- $\mathcal{V}_a[\![x.i]\!]_{e_a s_a} = \langle UB(\{s_a(r_n).i \mid n \geq i \wedge r_n \in e_a(x)\}), er \rangle$
  où $er = \begin{cases} \perp \text{ si } \forall\, y \in Ref_a \wedge y = r_n \wedge n \geq i \wedge s_a(y).i \notin noninit\, \forall\, y \in e_a(x) \\ error \text{ sinon} \end{cases}$
- $\mathcal{V}_a[\![this.i]\!]_{e_a s_a} = \langle v_{a_i}, er \rangle$
  où
  - $1 \leq i \leq n \wedge e_a(this) = r_n \wedge r_n \in Ref_a \wedge s_a(r_n) = \langle n, \langle v_{a_1}, .., v_{a_i}, .., v_{a_n} \rangle \rangle$
  - $er = \begin{cases} \perp \text{ si } i > 0 \wedge i \leq n \wedge e_a(this) \in Ref_a \\ error \text{ sinon} \end{cases}$

$\mathcal{A}op : Op \rightarrow \mathbb{V}al \rightarrow \mathbb{V}al \rightarrow \mathbb{V}al + \{error\}$
$\mathcal{A}op_a : Op \rightarrow \mathbb{V}al_a^f \rightarrow \mathbb{V}al_a^f \rightarrow \mathbb{V}al_a^f \times \{error, \perp\}$
$Val_a^f = Val_a \cap \mathbb{Z}_a$

Comportement :

$$\left.\begin{array}{lcl} \mathcal{A}op[\![op]\!]_{v_1 v_2} & = & v \\ \mathcal{A}op_a[\![op]\!]_{v_{a_1} v_{a_2}} & = & \langle v_a, er \rangle \end{array}\right\} v \in Cc(v_a)$$

$\mathcal{V}_a[\![sexpr1 \ op \ sexpr2]\!]_{e_a s_a} = \langle v_a, er \rangle$
Où :
- $\mathcal{V}_a[\![sexpr1]\!]_{e_a s_a} = \langle v'_{a_1}, er_1 \rangle$
- $\mathcal{V}_a[\![sexpr2]\!]_{e_a s_a} = \langle v'_{a_2}, er_2 \rangle$
- $v_{a_1} \triangleq v'_{a_1} \cap \mathbb{Z}_a$

– $v_{a_2} \triangleq v'_{a_2} \cap \mathbb{Z}_a$

Nous avons à présent filtré les valeurs abstraites résultant des évaluations de expr1 et de expr2 aux valeurs abstrayant un entier. Nous pouvons alors aisément dire dans quel cas une erreur peut se produire :
$er_f = \bot$ sauf...
$v_{a_1} \neq v'_{a_1} \Rightarrow er_f = error$
$v_{a_2} \neq v'_{a_2} \Rightarrow er_f = error$

$$\mathcal{A}op_a[\![op]\!]_{v_{a_1} v_{a_2}} = \langle v_a, er_{aop} \rangle \; er = \begin{cases} \bot \text{ si } er_1 = er_2 = er_{aop} = er_f = \bot \\ error \text{ sinon} \end{cases}$$

Définition de $\mathcal{A}op_a$:

+ (v_{a1} \ v_{a2})	{-}	{0,-}	{0}	{0,+}
{-}	{{-}, $\bot$}	{{-}, $\bot$}	{{-}, $\bot$}	{{0,-,+}, $\bot$}
{0,-}	{{-}, $\bot$}	{{0,-}, $\bot$}	{{0,-}, $\bot$}	{{0,-,+}, $\bot$}
{0}	{{-}, $\bot$}	{{0,-}, $\bot$}	{{0}, $\bot$}	{{0,+}, $\bot$}
{0,+}	{{0,-,+}, $\bot$}	{{0,-,+}, $\bot$}	{{0,+}, $\bot$}	{{0,+}, $\bot$}
{+}	{{0,-,+}, $\bot$}	{{0,-,+}, $\bot$}	{{+}, $\bot$}	{{+}, $\bot$}
{-,+}	{{0,-,+}, $\bot$}	{{0,-,+}, $\bot$}	{{-,+}, $\bot$}	{{0,-,+}, $\bot$}
{0,-,+}	{{0,-,+}, $\bot$}	{{0,-,+}, $\bot$}	{{0,-,+}, $\bot$}	{{0,-,+}, $\bot$}
{}	{$\bot$, *error*}	{$\bot$, *error*}	{$\bot$, *error*}	{$\bot$, *error*}

FIGURE 2.2 – $\mathcal{A}op_a[\![+]\!] v_{a1} v_{a2}$ (partie 1)

+ (v_{a1} \ v_{a2})	{+}	{-,+}	{0,-,+}	{}
{-}	{{0,-,+}, $\bot$}	{{0,-,+}, $\bot$}	{{0,-,+}, $\bot$}	{$\bot$, *error*}
{0,-}	{{0,-,+}, $\bot$}	{{0,-,+}, $\bot$}	{{0,-,+}, $\bot$}	{$\bot$, *error*}
{0}	{{+}, $\bot$}	{{-,+}, $\bot$}	{{0,-,+}, $\bot$}	{$\bot$, *error*}
{0,+}	{{+}, $\bot$}	{{0,-,+}, $\bot$}	{{0,-,+}, $\bot$}	{$\bot$, *error*}
{+}	{{+}, $\bot$}	{{0,-,+}, $\bot$}	{{0,-,+}, $\bot$}	{$\bot$, *error*}
{-,+}	{{0,-,+}, $\bot$}	{{0,-,+}, $\bot$}	{{0,-,+}, $\bot$}	{$\bot$, *error*}
{0,-,+}	{{0,-,+}, $\bot$}	{{0,-,+}, $\bot$}	{{0,-,+}, $\bot$}	{$\bot$, *error*}
{}	{$\bot$, *error*}	{$\bot$, *error*}	{$\bot$, *error*}	{$\bot$, *error*}

FIGURE 2.3 – $\mathcal{A}op_a[\![+]\!] v_{a1} v_{a2}$ (partie 2)

-	{-}	{0,-}	{0}	{0,+}	v_{a2}
{-}	{{0,-,+}, $\perp$}	{{0,-,+}, $\perp$}	{{-}, $\perp$}	{{-}, $\perp$}	
{0,-}	{{0,-,+}, $\perp$}	{{0,-,+}, $\perp$}	{{0,-}, $\perp$}	{{0,-}, $\perp$}	
{0}	{{+}, $\perp$}	{{0,+}, $\perp$}	{{0}, $\perp$}	{{0,-}, $\perp$}	
{0,+}	{{+}, $\perp$}	{{0,+}, $\perp$}	{{0,+}, $\perp$}	{{0,-,+}, $\perp$}	
{+}	{{+}, $\perp$}	{{+}, $\perp$}	{{+}, $\perp$}	{{0,-,+}, $\perp$}	
{-,+}	{{0,-,+}, $\perp$}	{{0,-,+}, $\perp$}	{{-,+}, $\perp$}	{{0,-,+}, $\perp$}	
{0,-,+}	{{0,-,+}, $\perp$}	{{0,-,+}, $\perp$}	{{0,-,+}, $\perp$}	{{0,-,+}, $\perp$}	
{}	{$\perp$, *error*}	{$\perp$, *error*}	{$\perp$, *error*}	{$\perp$, *error*}	
v_{a1}					

FIGURE 2.4 – $\mathcal{A}op_a[\![-]\!]v_{a1}v_{a2}$ (partie 1)

-	{+}	{-,+}	{0,-,+}	{}	v_{a2}
{-}	{{-}, $\perp$}	{{0,-,+}, $\perp$}	{{0,-,+}, $\perp$}	{$\perp$, *error*}	
{0,-}	{{-}, $\perp$}	{{0,-,+}, $\perp$}	{{0,-,+}, $\perp$}	{$\perp$, *error*}	
{0}	{{-}, $\perp$}	{{-,+}, $\perp$}	{{0,-,+}, $\perp$}	{$\perp$, *error*}	
{0,+}	{{0,-,+}, $\perp$}	{{0,-,+}, $\perp$}	{{0,-,+}, $\perp$}	{$\perp$, *error*}	
{+}	{{,0,-,+}, $\perp$}	{{0,-,+}, $\perp$}	{{0,-,+}, $\perp$}	{$\perp$, *error*}	
{-,+}	{{0,-,+}, $\perp$}	{{0,-,+}, $\perp$}	{{0,-,+}, $\perp$}	{$\perp$, *error*}	
{0,-,+}	{{0,-,+}, $\perp$}	{{0,-,+}, $\perp$}	{{0,-,+}, $\perp$}	{$\perp$, *error*}	
{}	{$\perp$, *error*}	{$\perp$, *error*}	{$\perp$, *error*}	{$\perp$, *error*}	
v_{a1}					

FIGURE 2.5 – $\mathcal{A}op_a[\![-]\!]v_{a1}v_{a2}$ (partie 2)

*	{-}	{0,-}	{0}	{0,+}	v_{a2}
{-}	{{+}, $\perp$}	{{0,+}, $\perp$}	{{0}, $\perp$}	{{0,-}, $\perp$}	
{0,-}	{{0,+}, $\perp$}	{{0,+}, $\perp$}	{{0}, $\perp$}	{{0,-}, $\perp$}	
{0}	{{0}, $\perp$}	{{0}, $\perp$}	{{0}, $\perp$}	{{0}, $\perp$}	
{0,+}	{{0,-}, $\perp$}	{{0,-}, $\perp$}	{{0}, $\perp$}	{{0,+}, $\perp$}	
{+}	{{-}, $\perp$}	{{0,-}, $\perp$}	{{0}, $\perp$}	{{0,+}, $\perp$}	
{-,+}	{{-,+}, $\perp$}	{{0,-,+}, $\perp$}	{{0}, $\perp$}	{{0,-,+}, $\perp$}	
{0,-,+}	{{0,-,+}, $\perp$}	{{0,-,+}, $\perp$}	{{0}, $\perp$}	{{0,-,+}, $\perp$}	
{}	{$\perp$, *error*}	{$\perp$, *error*}	{$\perp$, *error*}	{$\perp$, *error*}	
v_{a1}					

FIGURE 2.6 – $\mathcal{A}op_a[\![*]\!]v_{a1}v_{a2}$ (partie 1)

*	{+}	{-,+}	{0,-,+}	{}	v_{a2}
{-}	{{-}, ⊥}	{{-,+}, ⊥}	{{0,-,+}, ⊥}	{⊥, *error*}	
{0,-}	{{0,-}, ⊥}	{{0,-,+}, ⊥}	{{0,-,+}, ⊥}	{⊥, *error*}	
{0}	{{0}, ⊥}	{{0}, ⊥}	{{0}, ⊥}	{⊥, *error*}	
{0,+}	{{0,+}, ⊥}	{{0,-,+}, ⊥}	{{0,-,+}, ⊥}	{⊥, *error*}	
{+}	{{+}, ⊥}	{{-,+}, ⊥}	{{0,-,+}, ⊥}	{⊥, *error*}	
{-,+}	{{-,+}, ⊥}	{{-,+}, ⊥}	{{0,-,+}, ⊥}	{⊥, *error*}	
{0,-,+}	{{0,-,+}, ⊥}	{{0,-,+}, ⊥}	{{0,-,+}, ⊥}	{⊥, *error*}	
{}	{⊥, *error*}	{⊥, *error*}	{⊥, *error*}	{⊥, *error*}	
v_{a1}					

FIGURE 2.7 – $\mathcal{A}op_a[\![*]\!]v_{a1}v_{a2}$ (partie 2)

/,%	{-}	{0,-}	{0}	{0,+}	v_{a2}
{-}	{{+}, ⊥}	{{+}, *er*}	{{}, *er*}	{{-}, *er*}	
{0,-}	{{0,+}, ⊥}	{{0,+}, *er*}	{{}, *er*}	{{0,-}, *er*}	
{0}	{{0}, ⊥}	{{0}, *er*}	{{}, *er*}	{{0}, *er*}	
{0,+}	{{0,-}, ⊥}	{{0,-}, *er*}	{{}, *er*}	{{0,+}, *er*}	
{+}	{{-}, ⊥}	{{-}, *er*}	{{}, *er*}	{{+}, *er*}	
{-,+}	{{-,+}, ⊥}	{{-,+}, *er*}	{{}, *er*}	{{-,+}, *er*}	
{0,-,+}	{{0,-,+}, ⊥}	{{0,-,+}, *er*}	{{}, *er*}	{{0,-,+}, *er*}	
{}	{⊥, *error*}	{⊥, *error*}	{⊥, *error*}	{⊥, *error*}	
v_{a1}					

FIGURE 2.8 – $\mathcal{A}op_a[\![/]\!]v_{a1}v_{a2} \wedge \mathcal{A}op_a[\![\%]\!]v_{a1}v_{a2}$ où $er = error$ (partie 1)

/,%	{+}	{-,+}	{0,-,+}	{}	v_{a2}
{-}	{{-}, ⊥}	{{-,+}, ⊥}	{{-,+}, *error*}	{⊥, *error*}	
{0,-}	{{0,-}, ⊥}	{{0,-,+}, ⊥}	{{0,-,+}, *error*}	{⊥, *error*}	
{0}	{{0}, ⊥}	{{0}, ⊥}	{{0}, *error*}	{⊥, *error*}	
{0,+}	{{0,+}, ⊥}	{{0,-,+}, ⊥}	{{0,-,+}, *error*}	{⊥, *error*}	
{+}	{{+}, ⊥}	{{-,+}, ⊥}	{{-,+}, *error*}	{⊥, *error*}	
{-,+}	{{-,+}, ⊥}	{{-,+}, ⊥}	{{-,+}, *error*}	{⊥, *error*}	
{0,-,+}	{{0,-,+}, ⊥}	{{0,-,+}, ⊥}	{{0,-,+}, *error*}	{⊥, *error*}	
{}	{⊥, *error*}	{⊥, *error*}	{⊥, *error*}	{⊥, *error*}	
v_{a1}					

FIGURE 2.9 – $\mathcal{A}op_a[\![/]\!]v_{a1}v_{a2} \wedge \mathcal{A}op_a[\![\%]\!]v_{a1}v_{a2}$ (partie 2)

2.2.4 Affectations ($\mathcal{A}_a$)

On a

- $ass ::= \quad des := expr \quad | \quad x := new/i$
- $\mathcal{A} : Ass \to \mathbb{E}nv \to \mathbb{S}tore \to \mathbb{D} + \{error\}$

Soit la fonction $\mathcal{A}_a : Ass \rightarrow \mathbb{E}nv_a \rightarrow \mathbb{S}tore_a \rightarrow \mathbb{D}_a \times \{\perp, error\}$
$\mathcal{A}_a[\![x := expr]\!]_{e_a s_a} = \langle e'_a, s_a, er\rangle$
où :
- $\mathcal{V}_a[\![expr]\!]_{e_a s_a} = \langle v_a, er_{expr}\rangle$
- $\mathcal{D}_a[\![x]\!]_{e_a s_a} = \langle d_a, er_{des}\rangle$ avec $er_{des} = \perp$
- $e'_a = e_a[d_a/v_a]$
- $er = \begin{cases} \perp \text{ si } er_{des} = er_{expr} = \perp \\ error \text{ sinon} \end{cases}$

$\mathcal{A}_a[\![x.i = expr]\!]_{e_a s_a} = \langle e_a, s'_a, er\rangle$
où :
- $\mathcal{V}_a[\![expr]\!]_{e_a s_a} = \langle v_a, er_{expr}\rangle$
- $\mathcal{D}_a[\![x.i]\!]_{e_a s_a} = \langle d_a, er_{des}\rangle$
- $s'_a = s_a[d_a/v_a]$
- $er = \begin{cases} \perp \text{ si } er_{des} = er_{expr} = \perp \\ error \text{ sinon} \end{cases}$

$\mathcal{A}_a[\![this.i = expr]\!]_{e_a s_a} = \langle e_a, s'_a, er\rangle$
où :
- $\mathcal{V}_a[\![expr]\!]_{e_a s_a} = \langle v_a, er_{expr}\rangle$
- $\mathcal{D}_a[\![this.i]\!]_{e_a s_a} = \langle d_a, er_{des}\rangle$
- $s'_a = s_a[d_a/v_a]$
- $er = \begin{cases} \perp \text{ si } er_{des} = er_{expr} = \perp \\ error \text{ sinon} \end{cases}$

Soit la fonction alloc : $\mathbb{N} \rightarrow \mathbb{S}tore_a \rightarrow \mathbb{R}ef_a \times \mathbb{S}tore_a$
$alloc(i, s) = (r_i, UB(s, (r_i \rightarrow \langle i, \langle v_1, ...v_i\rangle\rangle)))$
avec $\langle v_1, ..., v_i\rangle = \{noninit\}$

$\mathcal{A}_a[\![x := new/i]\!]_{e_a s_a} = \langle e[x/r_i], s'\rangle$ si $i \geq 0$ avec $(r_i, s') = alloc(i, s)$

Pour chaque affectation, $er = \begin{cases} \perp \text{ si } er_{expr} = \perp \wedge er_{des} = \perp \\ error \text{ sinon} \end{cases}$

La notation $s[\langle r, i\rangle/v]$ signifie que la ou les valeurs du $i^{ème}$ champs de ou des objets se situant à l'adresse r est ou sont mis(e) à v.

2.3 Relations de transition

2.3.1 Affectations

- $\langle l, e_a, s_a, p_a\rangle \xrightarrow{l\ x := expr\ l_a} \langle l', e'_a, s_a, p_a\rangle$
  $l' = l_a$

$e'_a = \mathcal{A}[\![x := expr]\!]_{e_a s_a}$

- $\langle l, e_a, s_a, p_a\rangle \xrightarrow{l\ x.i\ :=\ expr\ l_a} \langle l', e_a, s'_a, p_a\rangle$
  $l' = l_a$
  $s'_a = \mathcal{A}[\![x.i := expr]\!]_{e_a s_a}$

- $\langle l, e_a, s_a, p_a\rangle \xrightarrow{l\ this.i\ :=\ expr\ l_a} \langle l', e_a, s'_a, p_a\rangle$
  $l' = l_a$
  $s'_a = \mathcal{A}[\![this.i := expr]\!]_{e_a s_a}$

- $\langle l, e_a, s_a, p_a\rangle \xrightarrow{l\ x\ :=\ new/i\ l_a} \langle l', e'_a, s'_a, p_a\rangle$
  $l' = l_a$
  $(e'_a, s'_a) = \mathcal{A}[\![x := new/i]\!]_{e_a s_a}$

2.3.2 Instruction *if*

Soit $B_a[\![cond]\!]_{e_a s_a} = \langle PD_{a_t}, PD_{a_f}, er\rangle$
$\langle l, e, s, p\rangle \xrightarrow{l\ if\ cond\ then\ l_1\ else\ l_2} \{\langle l_1, e', s', p\rangle \mid \langle e', s'\rangle \in PD_{a_t}\} \cup \{\langle l_2, e', s', p\rangle \mid \langle e', s'\rangle \in PD_{a_f}\}$

Si $\mid PD_{a_t} \mid = 0 \wedge \mid PD_{a_f} \mid = 0$, alors il n'y a pas de transition abstraite .

2.3.3 Appels de méthodes statiques

$$\langle l, e_a, s_a, p_a\rangle \xrightarrow{l\ \ x = m_c(y_{a1},...,y_{an})\ l''} \langle l', e'_a, s_a, p'_a\rangle$$

Méthode choisie : $m_c(z_{a1}, ..., z_{an})\ l_0\ stmt\ l_f\ x_f$
$l' = l_0$
$e'_a = (z_{ai} = \mathcal{V}[\![y_{ai}]\!]_{e_a s_a}) \quad \forall\, i : 1 \leq i \leq n$
p'_a est la pile p_a où on a ajouté abstraitement $\langle l'', e_a, x\rangle$.

Précisions sur p'_a (sur base de la section sur P_a)

Soit
- s_m est l'état (dans la machine à états de la pile) de la méthode dans laquelle on se trouve et dans lequel se trouve l'automate dans p_a
- s_{m_c} est l'état (dans la machine à états de la pile) correspondant à la méthode m_c

p'_a est alors p_a où l'automate est passé dans s_{m_c} par une transition t.

Définissons t :
- Soit $\exists\, t' = \langle s_m, \langle l'', e_a, x\rangle, s_{m_c}\rangle \in T$, alors $t = t'$

– Soit $\exists\, t' = \langle s_m, \langle l'', e_{trans}, x\rangle, s_{m_c}\rangle \in T$ où $e_{trans} \in \mathbb{E}nv_a$ et $e_{trans} \neq e_a$, alors t' est remplacée dans T par $t = \langle s_m, \langle l'', e'_{trans}, x\rangle, s_{m_c}\rangle$ où $e'_{trans} = UB(e_a, e_{trans})$.
– Soit $\nexists t' = \langle s_m, \langle l'', e_a, x\rangle, s_{m_c}\rangle \in T$, alors t' est ajoutée à T, $t = t'$, et toute autre transition entre s_m et s_{mc} signifiant un appel de méthode est supprimée.

2.3.4 Appels de méthodes dynamiques

Soient
– $X = \mathcal{V}_a[\![x]\!]_{e_a s_a}$, toutes les valeurs de x.
– $X' = X \cap \mathbb{R}ef_a$, l'ensemble des références de x.

Soit M_c, l'ensemble $m_{c_1}, \ldots, m_{c_m}$ qui ont pour identificateur le nom m_c et tel que : $\forall\, m_{c_j} \in M_c, \exists\, r_i \in X'$ tel que m_{c_j} soit la méthode choisie lors de l'appel $r_i.m_c$ avec $j \leq i \wedge \forall\, k \leq i, k \neq j \Rightarrow k < j$.

$\langle l, e_a, s_a, p_a\rangle \xrightarrow{l\ \ rf = x.m_c(y_{a_1}, \ldots, y_{a_n})\ l''} \{\langle l'_j, e'_{aj}, s_a, p'_{aj}\rangle \mid$
avec m_{cj} méthode de type $m_c/i(z_1 j, ..., z_k j)\ l_{0j}\ stmt_j\ lf_j\ xf_j$
$l'_j = l_{0_j}$
$e'_{aj} = (z_{a_j} = \mathcal{V}[\![y_{a_w}]\!]_{e_a s_a}) \quad \forall\, w : 1 \leq w \leq n \wedge \forall\, ri \in X'$
p'_{aj} est la pile p_a où on a ajouté abstraitement $\langle l'', e'_{aj}, rf\rangle$.
$\}_{\forall\, m_{cj} \in M_c}$

2.3.5 Appels vers super

$\langle l, e_a, s_a, p_a\rangle \xrightarrow{l\ \ x = super.m_c(y_{a_1}, \ldots, y_{a_n})\ l''} \langle l', e'_a, s_a, p'_a\rangle$
avec $m_c/i(z1, ..., zk)\ l0\ stmt\ l1\ xf$ la méthode choisie lors de l'appel à super tel que :
$k =$ au niveau de la methode courante.
$i < k \wedge \forall\, j < k, j \neq i \Rightarrow j < i$
$l' = l0$
$e'_a = (z_{a_i} = \mathcal{V}[\![y_{a_i}]\!]_{e_a s_a}) \quad \forall\, i : 1 \leq i \leq n$
soit m, la méthode courante auquelle l appartient.
p'_a est la pile p_a où on a ajouté abstraitement $\langle l'', e_a, x\rangle$.

2.3.6 Retour de méthode

Soient
– m_c la méthode dans laquelle on est
– m la méthode dans laquelle on revient

- la fonction $\mathcal{A}pp : S \rightarrow \mathcal{P}(\mathbb{L}abel \times \mathbb{E}nv_a \times \mathbb{X})$ qui renvoit tous les triplets $\langle l, e_a, x\rangle$ de toutes les transitions $\langle s_m, \langle l, e_a, x\rangle, s_{m_c}\rangle \in T$ où s_{m_c} est le paramètre passé à la fonction.
- p_a, la pile abstraite actuelle
- $Return(p, s)$ renvoie la pile abstraite p où l'automate est passé dans l'état s.

$$\begin{aligned}\langle l, e_a, s_a, p_a\rangle \longrightarrow \{&\langle l_{app}, e_{app}[x_{app}/\, V_a[\![x_f]\!]_{e_a s_a}], s_a, Return(p_a, s_m)\rangle \\ &\mid\ \langle l_{app}, e_{app}, x_{app}\rangle \in App_{s_{m_c}}\}\end{aligned}$$

Chapitre 3

Implémentation de l'interpréteur abstrait

3.1 Domaine abstrait - class AbstractDomain

Un domaine abstrait est un objet composé d'un environnement abstrait et d'un store abstrait (classes EnvA et StoreA).

Les opérations possibles sont :

- void addVal(Identifier i, ValA va) : ajoute une nouvelle valeur dans l'environnement.
  préconditions : i et va sont non null.
  postconditions : le couplet (i $\rightarrow$ va) est ajouté dans l'environnement (écrase le précédent s'il en existe un).

- ValA getValA(Identifier i) : permet de récupérer une valeur dans l'environnement abstrait.
  préconditions : i est non null.
  postconditions : i est inchangé.
  return : la valeur abstraite correspondant à l'identificateur i si i est présent dans l'environnement abstrait.
  return : une valeur abstraite correspondant à noninit sinon.

- ObjA getObjA(RefA ref) : permet de retrouver un objet abstrait du store abstrait suivant une référence abstraite.
  préconditions : ref est non null.
  postconditions : ref est inchangé.
  return : l'objet abstrait correspondant à ref s'il y a une telle référence dans le store abstrait.
  return : null sinon.

- void addObjA(RefA refA, ObjA objA) : permet d'ajouter un nouvel objet au store.
  préconditions : refA et objA sont non null.
  postconditions : refA et objA sont inchangés.
  postconditions : le couple (refA→objA) est ajouté au store abstrait s'il n'en existe pas un,
  le couple (refA→objA.ub(getObjA(refA))) est ajouté au store abstrait sinon.

- void setObjetField(RefA refA, Integer i, ValA vA) : permet de modifier un champ d'un objet abstrait.
  préconditions : refA, i et vA sont non null.
  postconditions : refA, i et vA sont inchangés.
  postconditions : le champ i de l'objet correspondant à refA est mis à jour avec la valeur de vA.ub(getObjA(refA)).

- boolean leq(AbstractDomain da) calcule la relation inférieure ou égale de la relation bien fondée de l'algorithme polyvariant.
  préconditions : da est non null.
  postconditions : da eest inchangé.
  return : true si this est inférieur ou égal à da suivant la relation bien fondée,
  false sinon.

- AbstractDomain ub(AbstractDomain dA) calcul la borne supérieure de deux domaines abstraites.
  préconditions : dA est non null.
  postconditions : dA est inchangé.
  return la borne supérieur de this et dA.

Ces méthodes font appels aux mêmes méthodes des class EnvA et/ou StoreA.

3.1.1 Environnement et Store abstraits - class EnvA et class StoreA

Ces deux structures de données (ainsi que la pile abstraite) sont construites au moyen d'une **architecture en couches**. Nous avons beaucoup réfléchi sur ce point afin d'alléger les ressources mémoires. Notre but était d'éviter au maximum de devoir cloner des structures de données telles quelles.

Pourtant, il est important de voir que l'algorithme polyvariant demande de sauvegarder des anciens environnements et stores notamment lors d'un branchement conditionnel, afin de pouvoir interpréter la branche *true* puis de revenir au point de programme qui précède le branchement, avec l'environnement et le store de ce point de programme, dans le but d'interpréter la branche *false*.

L'idée de départ est que chaque environnement (store) est une liste de sous-environnements (sous-stores) où chaque élément de cette liste est un dictionnaire effectuant les correspondances identifiant-valeur (reference-objet). Dans cette liste, le dernier élément constitue la première couche (la plus ancienne) et le premier élément la dernière couche (la plus récemment ajoutée).

Un invariant de classe de ces structures de données devrait mentionner clairement que seul le premier dictionnaire de la liste (la couche la plus récente) est modifiable. Les autres éléments sont uniquement accessibles en lecture.

Concrètement :

– Lorsque l'on veut ajouter une paire identifiant-valeur, celle-ci sera toujours ajoutée dans le dictionnaire constituant la dernière couche (la plus récente). Cette opération est donc en $O(1)$.
– Lorsque l'on veut récupérer une valeur pour un identifiant, l'algorithme va d'abord aller voir dans la dernière couche (la plus récente) et descendra de couches en couches jusqu'à obtenir la valeur recherchée. Si la valeur ne se trouve dans aucune couche, c'est qu'on est dans un cas d'erreur. Cette opération est donc en $O(n)$ où n est le nombre de couches.
– Lorsque l'on veut sauvegarder un environnement ou un store avec n couches, il faudra simplement créer un environnement ou un store de n + 1 couches, où la plus récente est un dictionnaire vide et les autres sont des pointeurs vers les dictionnaires à sauvegarder. Il s'agit donc de créer une extension de l'environnement ou du store pour les exécutions futures. L'utilisation de pointeurs explique la gain de ressources utilisées, l'information n'étant pas dupliquée.

Pour illustrer ce mécanisme, prenons l'exemple d'un programme simple de trois instructions :

1. `a=3-2; b=0;`
2. `if(a==0) a=1; else a=-1;`
3. `write(a); write(b);`

L'algorithme polyvariant devra dans cet exemple interpréter la branche *true* et la branche *false* car avant le branchement la valeur de a résultant de l'opération $\{+\} - \{+\}$ est $\{0, -, +\}$.

Explications :

– La première instruction aura un environnement composé d'une seule couche C1 avec les couples $\langle\{a, \{0, -, +\}\}\rangle$ et $\langle\{b, \{0\}\}\rangle$.
– La deuxième instruction pour le cas *true* aura un environnement de deux couches, la plus ancienne étant un pointeur vers C1 et la plus

récente contiendra un couple $\langle\{a, \{+\}\}\rangle$. La troisième instruction affichera $\{+\}$ pour a car l'algorithme regarde d'abord la couche plus récente et $\{0\}$ pour b (l'algorithme n'ayant rien trouvé pour b dans la couche la plus récente, il aura regardé dans la couche d'en-dessous)
– La deuxième instruction pour le cas *false* aura un environnement de deux couches, la plus ancienne étant un pointeur vers C1 et la plus récente contiendra un couple $\langle\{a, \{-\}\}\rangle$. La troisième instruction affichera $\{-\}$ pour a et $\{0\}$ pour b.

Dans cet exemple, on peut voir que le couple $\langle\{b, \{0\}\}\rangle$ ne se trouve que dans une seule structure de données et qu'on évite des copies inutiles d'informations. Dans des exemples plus complexes, le dédoublement d'exécutions avec la sauvegarde d'environnement et store pourra être assez élevé, ce qui montrera davantage à quel point une architecture en couches est importante.

Afin d'être complets, il faut aussi admettre quelques désavantages comme la récupération d'un élément qui peut demander dans le pire cas de parcourir toutes les couches ou encore une utilisation abusive de dictionnaires qui sont des structures de données relativement lourdes mais très utiles pour avoir un accès en $O(1)$ attendu.

Une alternative étudiée et rejetée consistait à attribuer un numéro de version à chaque instruction et ainsi n'avoir qu'une et une seule grosse structure de données. La vérification des numéros de version lors de la récupération d'une valeur aurait été trop lourde dans un programme complexe. De plus, une architecture en couches nous paraissait plus logique lorsque nous avons dessiné sur papier l'arbre d'exécutions.

3.2 Les objets abstraits - class ObjA

Un objet abstrait est représenté par une référence qui l'identifie ainsi qu'un tableau de valeurs abstraites.
Les méthodes publiques appliquables à un objet abstrait sont :

- ValA getField(int i) renvoie le champs i d'un objet abstrait.
  préconditions : i est inférieur ou égale à la valeur fields[0] c-à-d le nombre de champs de l'objet abstrait.
  return la valeur abstraite du champs i-1 de l'objet.

- setField(int i, ValA va) met à jour un champs d'un objet abstrait.
  préconditions : i est inférieur ou égale à la valeur fields[0], va est non null.
  postconditions : va est inchangé.
  postconditions : le champs i de l'objet abstrait est mis à jour a la valeur va.

– int getRef() la référence qui identifie l'objet abstrait.

– boolean leq(ObjA obja, StoreA sA1, StoreA sA2) calcul test si un objet est inférieur ou égale à un autre objet suivant la relation bien fondée.
préconditions : obja, sA1, sA2 sont non null.
préconditions : sA1 est le store abstrait associé à this, et sA2 est le store abstrait associé a obja.
postconditions : obja, sA1, sA2 sont inchangés.
return true si this est inférieur ou égale à obja suivant la rélation bien fondée. false sinon.

– ObjA ub(ObjA obja) calcul la borne supérieur de deux objets abstraits.
préconditions : obja est non null.
préconditions : obja et this ont la même référence (cette précondition est testé dans la méthode)
postconditions : obja est inchangé.
return un objet abstrait qui est la borne supérieur de this et obja.

– ObjA intersection(ObjA obja) calcul l'intersection de deux objets abstraits.
préconditions : obja est non null.
préconditions : obja et this ont la même référence
postconditions : obja est inchangé.
return un objet abstrait qui est l'intersection de this et obja.

– ObjA except(ObjA obja) calcul un sous-ensemble d'un objet abstrait.
préconditions : obja est non null.
préconditions : obja et this ont la même référence
postconditions : obja est inchangé.
return un objet abstrait correspondant aux valeurs de this otée de tout les valeurs de obja.

3.3 Valeurs abstraites - class ValA

Une valeur abstraite est toute combinaison de valeurs possibles des éléments de l'ensemble : {valeur entière abstraite, un ensemble de références, null , noninit }. Une valeur entière abstraite est représentée par un entier défini dans la class PZa, l'ensemble des références est représenté par un objet de la class PRefA, null et noninit sont chacun représenter par un booléen.
Les méthodes applicables à une valeur abstraite sont :

– tester si une valeur abstraite est contenue dans celle-ci (par exemple :

tester si null fait partie des valeurs d'une variable.)
– mettre à jour la valeur abstraite (ses valeurs de base)
– calculer une borne supérieure, l'intersection, la relation inférieure ou égale de la relation bien fondée entre deux valeurs abstraites.

3.3.1 Le domaine des entiers - class PZa

Une valeur entière abstraite est représenté par une combinaison possible des éléments de l'ensemble {-,0,+} et donc toute valeur entière abstraite ne peut-être que l'une des huit cas de figure ci-dessous :

– composé d'un ensemble vide, représenté par la valeur EMPTY.
– composé de l'unique élément '-' représenté par la valeur MINUS.
– composé des éléments '-' et '0' représenté par la valeur MZ.
– composé de l'unique élément '0' représenté par la valeur ZERO.
– composé des éléments '0' et '+' représenté par la valeur ZP.
– composé de l'unique élément '+' représenté par la valeur PLUS.
– composé des éléments '-' et '+' représenté par la valeur MP.
– composé des éléments '-', '0' et '+' représenté par la valeur MZP.

Ces éléments sont en fait des entiers accessibles au moyen de champs static final. Toute les opérations sur ces valeurs sont précalculés et stockés sous forme de tableaux afin d'avoir des accès en $O(1)$ pour chaque opération.

Les opérations applicables sur une ou deux valeurs entières abstraites sont :
– ub(int i, int j) : qui calcule la borne supérieur des deux éléments et qui consiste à calculer l'union des deux ensembles.
– intersection(int i, int j) : qui renvoie un nouvel ensemble qui contient l'intersection des deux ensembles.
– except(int i, int j) : qui renvoie un nouvel ensemble contenant les éléments du premier ensemble sauf ceux qui sont présents dans le second ensemble.
– leq(int i, int j) : qui renvoie true si le premier ensemble est inclus dans le second. renvoie false sinon.
– contains(int i, int j)qui renvoie true si le premier ensemble contient le second ensemble. renvoie false sinon.
– containsVals(int i) renvoie true si l'ensemble ne correspond pas à l'ensemble vide.
– toString(int i) : renvoie un string de l'ensemble.
– getElements(int i) : renvoie l'ensemble représenté par un tableau de caractères
– op(int i, int j) : renvoie un nouvel ensemble qui est le résultat de l'opération plus arithmétique entre les deux ensembles.
(op $\in$ {plus,minus,times,div,mod})

- opErr(int i,int j) : renvoie true si il y'a une erreur dans l'opération op entre deux ensembles. renvoie false sinon.
  (op $\in$ {plus,minus,times,div,mod})

Le choix d'une telle implémentation se justifie par le fait qu'une valeur entière ne peut-être qu'une des 8 combinaisons de valeurs possibles et qu'a chaque opération, nous ne pouvons qu'obtenir les mêmes résultats pour deux ensembles donnés, cela nous permet d'éviter les calculs répétitifs, et un accès immédiat au résultat des opérations.

3.3.2 Les références - class RefA - class PRefA

Une référence abstraite est un objet identifié par un entier (class RefA), un ensemble de références abstraites est représenté par une liste de références abstraites (classPRefA), les opérations applicables à un ensemble sont :

- boolean containsVals() : vérifier si l'ensemble est vide.
- boolean contains(PRefA refa) : vérifier si un ensemble de références est inclus dans un autre ensemble de références.
- boolean boolean leq(PRefA refa, StoreA sA1, StoreA sA2) : calculer la relation inférieur ou égale de la relation bien fondée.

3.4 La pile abstraite - class StackA

Notre pile est aussi représentée avec une architecture en couches car les transitions sont ajoutées dynamiquement. Dans les branchements conditionnels, il arrive notamment que des appels de méthodes ne soient faits que dans le cas *true* par exemple. Il faut donc aussi sauvegarder des piles différentes pour les différentes exécutions, et à nouveau, notre choix a été l'architecture en couches.

Chaque couche de la pile est un dictionnaire à deux dimensions, qui sont la méthode appelante et la méthode appelée. Cela est très avantageux car...

- On peut obtenir ou ajouter directement en $O(1)$ une transition lorsque l'on connaît méthode appelante et méthode appelée. C'est le cas lors d'un appel de méthode.
- On peut obtenir toutes les transitions qui ont une méthode appelée fixée en $O(n)$ où n est le nombre de transitions résultantes. C'est très utile lors d'un retour de méthode.

3.4.1 Les états de la pile - class Method

Nous avons souhaité par soucis de simplicité utiliser directement la classe Method comme état de la pile. C'est aussi les instances de celle-ci qui sont les clés du dictionnaire à deux dimensions pour les transitions.

3.4.2 Les transitions dans la pile - class Transition

Une transition est caractérisée par sa méthode appelante, méthode de destination, la variable de retour, le premier label de la méthode appelée, un domaine abstrait de l'appel entre la méthode appelante et de la méthode appelée.
Les méthodes publiques de la class Transition nous permettent de récupérer les informations cités ci-dessus et de mettre à jour le domaine abstrait.

Les méthodes publiques applicables à une pile abstraite sont :

– void setTransition(Method fromNode, AbstractDomain dAFrom, Stmt nextStmt, Identifier returnVariable, Method toNode) ajoute ou met à jour une transition dans la pile abstraite.
préconditions : fromNode, dAFrom, nextStmt, returnVariable, toNode sont non null.
postconditions : fromNode, dAFrom, nextStmt, returnVariable, toNode sont inchangés.
postconditions : s'il n'existe pas de transition t tels que : t = (fromNode', dAFrom', nextStmt', returnVariable', toNode'), fromNode=fromNode', toNode=toNode' et tel que t $\notin$ transitionsList (la liste des transitions de la pile abstraite) alors une nouvelle transition contenant les paramètres précités est ajouté à transitionsList. sinon c'est qu'il existe une transition t tel que fromNode=fromNode' et toNode=toNode', le domaine abstrait de t est mis à jour avec la borne supérieur de dAFrom et dAFrom'.

– Collection<Transition> getTransitions(Method toNode) renvoie toutes les transitions dans laquelle toNode est la méthode appelée.
préconditions : toNode est non null.
postconditions : toNode est inchangé.
return une collection de transition ct tel que $\forall\, t \in ct, t = (\textit{methodeFrom} \rightarrow (\textit{methodeTo} \rightarrow \textit{transitionFromTo}))$ et methodeTo=toNode.

3.5 L'interpréteur - class Interpreter

3.5.1 Fonctionnement de l'interpréteur

Nous avons choisi l'algorithme polyvariant avec subsomption pour interpréter le code slip. Le pseudo-code du cours est retranscrit dans la méthode *PolyWithSubsomption()*. Au début, un état composé de la première instruction de la main, d'un domaine abstrait vierge et d'une pile vierge est ajouté à l'ensemble S. Ensuite commence l'algorithme à proprement parlé. Le plus vieil état est retiré de S et ajouté à R. De cet état, on va trouver un ou plusieurs autres états. L'exécution d'un *statement if* ou un appel de méthode provoquent la création d'état(s). Chaque état créé est alors comparé à ceux se trouvant dans R par la relation d'ordre $\leq$. S'il existe un état dans R strictement plus grand que celui trouvé, alors il n'est pas ajouté à S.

La classe *Interpreter* contient toutes les fonctions explicitées dans le rapport à savoir, pour les plus importantes :

- $\mathbb{V}$ qui renvoie l'ensemble des valeurs de base d'une expression de droite.
- $\mathbb{A}$ qui exécute un *statement* de type autre que condition pour un domaine abstrait et une pile définis. Elle renvoie un couple constitué du domaine abstrait résultant de l'exécution du *statement* et un *boolean* indiquant si oui ou non il y a eu une erreur. Dans le cas d'un appel de méthode, un nouvel état est créé (ou plusieurs si on a affaire à un appel de type *x.m()*). Cet état contiendra comme prochaine instruction soit la première instruction de la méthode à appeler soit l'instruction suivant l'appel. Le premier choix est utilisé si il n'existe aucune transition vers la méthode appelée ou si le domaine abstrait courant est $\leq$ au domaine abstrait de la transition.
- $\mathbb{B}$ qui exécute un *statement* de type condition dans un domaine abstrait et une pile définis. Elle renvoie simplement un *boolean* indiquant si oui ou non il y a eu une erreur. De plus, cette fonction ajoute, sous les conditions de la subsomption, des états dans S correspondant à des états à exécuter dans le cas d'une condition vraie ou fausse. Les domaines abstraits de ces états sont généralement raffinés grâce à des heuristiques. Dans le cas où les heuristiques ne sont pas en mesure de raffiner le domaine abstrait, seulement deux états sont ajoutés, un pour le cas vrai et un autre pour le cas faux avec le domaine abstrait inchangé.
- $\mathbb{A}ss$ qui assigne à une expression de type x, $x.i$ ou *this.i* des valeurs de bases de type entier. Cette fonction renvoie le domaine abstrait modifié.

Chaque *statement* de type commande, condition ou méthode peut donc être défini comme provoquant une erreur ou non. Un *statement* se verra attribuer une annotation ; *OK* si chacune de ses exécutions n'a pas provoqué d'erreur ; *KO* si au moins de ses exécutions à provoquer une erreur ; *NR* si ce *statement* n'a jamais été exécuté ; et *UK* pour le reste.

3.5.2 Attribution des annotations

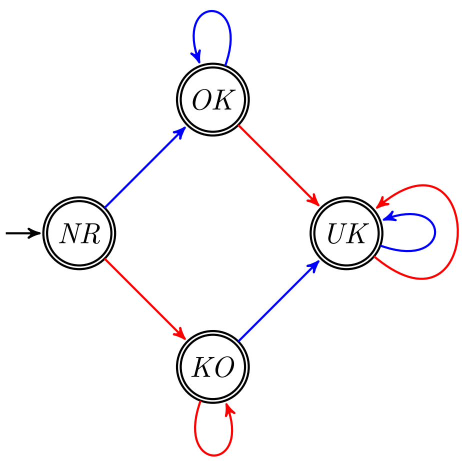
Chaque identifiant d'instruction (X dans `\\ X` dans le code fourni) représente un ensemble de *statements*. Afin de faciliter le travail tout en offrant une solution modulaire, nous avons opté pour un définition semi-automatique. C'est-à-dire qu'il est demandé à la personne souhaitant tester un fichier .slip de construire un fichier du même nom avec l'extension .properties dans le même répertoire avec à chaque ligne `"X = <x1, x2, ..., xn>"` où X est donc un identifiant d'instruction et x1, x2, ..., xn des numéros de labels dans le code interne. Notons que le X ne doit pas obligatoirement être un entier, ce qui peut être intéressant pour faciliter la lecture des résultats. Ce procédé est générique car il peut être appliqué sur n'importe quel fichier .slip mais demande l'implication du testeur pour créer ce fichier .properties.

L'attribution des annotations se fait en deux temps

1. Pendant l'interprétation, chaque *statement* peut recevoir une annotation, susceptible d'être modifiée plusieurs fois durant l'interprétation.
2. Après l'interprétation, les annotations pour chaque identifiant d'instruction sont attribuées sur base des annotations des *statements* qui le définisse.

Afin d'expliquer le fonctionnement de ces deux étapes, nous modélisons les attributions au moyen de machines à états. Grâce à cette modélisation, l'implémentation en Java a été simplifiée.

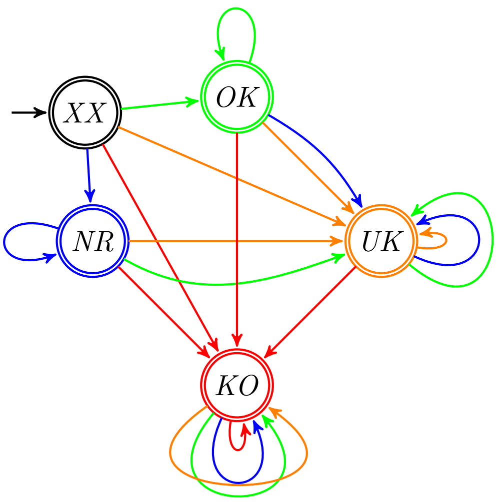
Machine à états pour attribuer une annotation à un *statement*



A l'initialisation, on a un tableau avec pour chaque *statement* la valeur NR. A chaque fois qu'un *statement* est exécuté avec un environnement et un store précis, on obtient *true* (flèche rouge) si l'exécution a provoqué une erreur et *false* (flèche bleue) sinon. Suivant l'annotation précédemment attribuée et le fait qu'il y ait eu erreur ou non, une (nouvelle) annotation est attribuée. Cela se modélise bien en une machine à états où l'on voit bien l'impossibilité de revenir à NR une fois qu'on l'a quitté mais aussi l'impossibilité de sortir de UK une fois qu'on est entré dedans !

Maintenant que chaque *statement* a une annotation, il faut renseigner une annotation pour chaque identifiant d'instruction. Ici, il s'agira donc de parcourir les annotations données à chaque *statement* d'un identifiant d'instruction. Il faudra alors attribuer une annotation à chaque identifiant d'instruction sur base de l'annotation donnée au *statement* courant et de l'annotation précédemment donnée à l'identifiant d'instruction.

Machine à états pour attribuer une annotation à un identifiant d'instruction



A nouveau, il est possible et plus simple de modéliser cela au moyen d'une machine à états. Dans celle-ci, la couleur des flèches indique l'annotation du *statement* courant. Comme on peut le voir, au départ chaque identifiant d'instruction est déclaré XX (pas de valeur). Pour que le résultat final soit NR (resp. OK), il faut impérativement que chaque *statement* de l'identifiant d'instruction ait eu l'annotation NR (resp. OK). On remarque aussi qu'il suffit qu'un seul des *statements* de l'identifiant d'instruction soit KO pour que l'annotation finale soit KO.

3.5.3 Mode d'emploi

Pour tester l'interpréteur, il suffit de passer en paramètre à la classe *Interpreter* le nom de fichier (ou chemin) du fichier slip à tester sans l'extension *.slip*. Par exemple : *java -cp bin ;ingi2132.jar Interpreter "chemin_complet pEP"*. Les fichiers *.ann* et *.urm* sont alors créés dans le même répertoire que le fichier slip testé. Le fichier *temps.txt* est créé dans le chemin courant de l'utilisateur. Le projet a été compilé avec la jdk 1.6, mais ne devrait pas poser de difficultés sous jdk 1.5.

3.6 Erreur détectée

Malheureusement, une erreur non résolue perturbe les résultats obtenus. En effet, les *statements* de type condition provoquent la création d'états à ajouter à S. Mais dans le cas où le domaine abstrait de ces états est $\leq$ à un des états de R, l'état n'est pas ajouté à S. On se retrouve donc, par exemple, avec des *statements* non atteignables alors que ces derniers devraient l'être. L'application se révèle particulièrement rapide peut-être en partie suite à ce problème qui doit limiter un certain nombre de tests.

3.7 Résultats

Les résultats des tests se trouvent dans le répertoire *tests* à la racine du projet. Il existe un répertoire pour chaque test et dans chacun de ces répertoires se trouvent les fichiers *.ann* et *.urm*. Nous n'avons pas ajouté les résultats au rapport afin de ne pas le surcharger. Excepté pour l'erreur détectée ci-dessus, les résultats retournés nous paraissent corrects après vérification manuelle.

3.8 Pistes d'améliorations

Nous avions dès le début pensé à l'idée de faire de l'exécution parallèle avec plusieurs processeurs de manière à améliorer l'efficacité. Une amélioration intéressante serait ainsi la mise en place de la concurrence. Enfin, il serait intéressant d'afficher des messages d'erreurs plus précis et plus complets à l'utilisateur qui le renseignent davantage sur l'erreur trouvée.