Analyse de Données Structurées - Cours 11

Ralf Treinen

Université Paris Diderot
UFR Informatique
Laboratoire Preuves, Programmes et Systèmes

treinen@pps.univ-paris-diderot.fr

15 avril 2015

Syntaxe Abstraite

L'ensemble E'' des arbres de syntaxe abstraite représentant des expressions arithmétiques et booléennes est défini comme suit :

- tout $Int(n)$, où $n \in \mathbb{N}$, est un élément de E''
- $True$ et $False$ sont des éléments de E''
- tout $Ident(s)$, où $s \in \Sigma^*$, est un élément de E''
- si $e_1, e_2 \in E''$, alors $Plus(e_1, e_2) \in E''$
- si $e_1, e_2 \in E''$, alors $Mult(e_1, e_2) \in E''$
- si $e_1, e_2 \in E''$, alors $Greater(e_1, e_2) \in E''$
- si $e_1, e_2 \in E''$, alors $Equal(e_1, e_2) \in E''$
- si $e_1, e_2 \in E''$, alors $And(e_1, e_2) \in E''$
- si $e_1, e_2 \in E''$, alors $Or(e_1, e_2) \in E''$
- si $e \in E''$, alors $Not(e) \in E''$

Règles de typage

- Environnement de typage : fonction partielle $\Sigma^* \to \{int, bool\}$
- Jugement $\Gamma \vdash e : t$: Dans l'environnement Γ, l'expression e a le type t.

$$\frac{}{\Gamma \vdash Int(n) : int} \quad n \in \mathbb{N}$$

$$\frac{}{\Gamma \vdash True : bool} \qquad \frac{}{\Gamma \vdash False : bool}$$

$$\frac{}{\Gamma \vdash Ident(s) : \Gamma(s)} \quad s \in \Sigma^*$$

Règles de typage

$$\frac{\Gamma \vdash e_1 : int \qquad \Gamma \vdash e_2 : int}{\Gamma \vdash Plus(e_1, e_2) : int}$$

$$\frac{\Gamma \vdash e_1 : int \qquad \Gamma \vdash e_2 : int}{\Gamma \vdash Mult(e_1, e_2) : int}$$

$$\frac{\Gamma \vdash e_1 : int \qquad \Gamma \vdash e_2 : int}{\Gamma \vdash Equal(e_1, e_2) : bool}$$

$$\frac{\Gamma \vdash e_1 : int \qquad \Gamma \vdash e_2 : int}{\Gamma \vdash Greater(e_1, e_2) : bool}$$

Règles de typage

$$\frac{\Gamma \vdash e_1 : bool \qquad \Gamma \vdash e_2 : bool}{\Gamma \vdash And(e_1, e_2) : bool}$$

$$\frac{\Gamma \vdash e_1 : bool \qquad \Gamma \vdash e_2 : bool}{\Gamma \vdash Or(e_1, e_2) : bool}$$

$$\frac{\Gamma \vdash e : bool}{\Gamma \vdash Not(e) : bool}$$

Implémentation : Expression.java (début) I

```
abstract class Expression {
    Type type;
    final public Type getType() {
        return this.type;
    }
    abstract void setType(TypeEnvironment env) throws TypeExceptio
}

class Int extends Expression {
    private int value;
    public Int (int i) {
        value=i;
    }
    public void setType(TypeEnvironment env) {
```


Implémentation : Expression.java (début) II

```
        this.type=Type.Int;
    }
}

class True extends Expression {
    public True () {
    }
    public void setType(TypeEnvironment env) {
        this.type=Type.Bool;
    }
}

class False extends Expression {
    public False () {
    }
```


Implémentation : Expression.java (début) III

```
    public void setType(TypeEnvironment env) {
        this.type=Type.Bool;
    }
}

class Ident extends Expression {
    private String name;
    public Ident (String s) {
        name=s;
    }
    public void setType(TypeEnvironment env) {
        this.type=env.get(name);
    }
}
```

Implémentation : Expression.java (début) IV

```
class Sum extends Expression {
    private Expression left;
    private Expression right;
    public Sum(Expression e1, Expression e2) {
        left=e1;
        right=e2;
    }
    public void setType(TypeEnvironment env) throws TypeException{
        left.setType(env);
        right.setType(env);
        if (left.getType()==Type.Int
            && right.getType()==Type.Int) {
            this.type=Type.Int;
        } else {
            throw new TypeException("Sum");
```


Implémentation : Expression.java (début) V

```
            }
        }
    }
```


Une représentation plus générique ?

- Cela devient assez fastidieux et répétitif.
- Imaginez qu'est-ce que ça donnera si on ajoute encore plus d'opérateurs, ou des nouvelles sortes (flottants, chaînes de caractère, ...)
- Solution plus compacte, et plus facile à généraliser :
  - Définir une sorte d'opérateurs (une par nombre d'arguments)
  - Définir un opérateur d'application d'un opérateur au bon nombre d'expressions.

Syntaxe abstraite générique : opérateurs et constantes

- L'ensemble O_2 des opérateurs en syntaxe abstraite est :

$$O_2 = \{Plus, Mult, Greater, Equal, And, Or\}$$

- L'ensemble O_1 des opérateurs unaires en syntaxe abstraite est

$$O_1 = \{Not\}$$

- L'ensemble O_0 des constantes en syntaxe abstraite est

$$O_0 = \{True, False\} \cup \{Int(n) \mid n \in \mathbb{N}\}$$

Syntaxe abstraite : expressions

L'ensemble E''' des expressions en syntaxe abstraite est :

- $Ident(s) \in E'''$ pour tout $s \in \Sigma^*$
- si $o \in O_0$ alors $Const(o) \in E'''$ pour tout $n \in \mathbb{N}$
- si $o \in O_1$, $e \in E'''$ alors $Apply_1(o, e) \in E'''$
- si $o \in O_2$, $e_1, e_2 \in E'''$ alors $Apply_2(o, e_1, e_2) \in E'''$

Typage pour la forme générique

- L'ensemble des types de base est $BT = \{int, bool\}$
- On va définir une fonction ot qui associe :
  - à toute constante $o \in O_0$ un élément de BT,
  - à tout opérateur unaire $o \in O_1$ une paire dans $BT \times BT$
  - à tout opérateur binaire $o \in O_2$ un triplet dans $BT \times BT \times BT$.
- Dans tous les cas, le dernier type est celui du résultat de l'application de l'opérateur, et les types précédents ceux des arguments.
- Parfois notation : $t_1 \times t_2 \to t_3$ au lieu de (t_1, t_2, t_3).

Typage des constantes et opérateurs

- Constantes :

$$\begin{aligned} ot(True) = ot(False) &= bool \\ ot(Int(n)) &= int \end{aligned}$$

- Opérateurs unaires :

$$ot(Not) = (bool, bool)$$

- Opérateurs binaires :

$$\begin{aligned} ot(Plus) = ot(Mult) &= (int, int, int) \\ ot(And) = ot(Or) &= (bool, bool, bool) \\ ot(Greater) = ot(Equal) &= (int, int, bool) \end{aligned}$$

Typage des expressions

$$\frac{}{\Gamma \vdash Ident(s) : \Gamma(s)} \qquad s \in \Sigma^*$$

$$\frac{}{\Gamma \vdash Const(o) : t} \qquad o \in O_0, ot(o) = t$$

$$\frac{\Gamma \vdash e_1 : t_1}{\Gamma \vdash Apply_1(o, e) : t} \qquad o \in O_1, ot(o) = (t_1, t)$$

$$\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash Apply_2(o, e_1, e_2) : t} \qquad o \in O_2, ot(o) = (t_1, t_2, t)$$

Syntaxe Abstraite

- Définitions mutuellement récursives des Instructions et des Listes d'Instructions.
- L'ensemble I des Instructions en syntaxe abstraite est :
  - si $e \in E'''$ alors $Print(e) \in I$,
  - si $s \in \Sigma^*, e \in E'''$ alors $Assign(s, e) \in I$,
  - si $e \in E''', il_1, il_2 \in IL$ alors $Cond(e, il_1, il_2) \in I$,
  - si $e \in E''', il \in IL$ alors $While(e, il) \in I$.
- L'ensemble IL des Listes d'Instructions en syntaxe abstraite est :
  - $Nil \in IL$,
  - si $i \in I$ et $il \in IL$ alors $Seq(i, il) \in IL$.

Syntaxe Abstraite

- L'ensemble D des Déclaration en syntaxe abstraite est :
  - si $s \in \Sigma^*$ et $t \in BT$ alors $Decl(s, t) \in D$.
- L'ensemble DL des Listes de Déclarations en syntaxe abstraite est :
  - $DeclNil \in DL$,
  - si $d \in D$ et $dl \in DL$ alors $DeclSeq(d, dl) \in DL$.
- L'ensemble P des Programmes en syntaxe abstraite est :
  - si $dl \in DL$ et $il \in IL$ alors $Program(dl, il) \in P$.

Déclarations et typage

- Jugement $dl : \Gamma$: la liste de déclaration dl donne l'environnement de typage Γ.
- Règles :

$$\frac{}{DeclNil : \emptyset}$$

$$\frac{dl : \Gamma}{DeclSeq(Decl(s, t), dl) : \Gamma[s \mapsto t]} \quad \text{si } s \notin \text{domaine}(\Gamma)$$

Implémentation : `DeclarationList.java` I

```
abstract class DeclarationList {
    abstract TypeEnvironment extract() throws DeclException;
}

class DeclNil extends DeclarationList {
    public DeclNil() {
    }
    public TypeEnvironment extract() {
        return new TypeEnvironment();
    }
}

class DeclSeq extends DeclarationList {
    private Declaration head;
```

Implémentation : DeclarationList.java II

```
private DeclarationList rest;
public DeclSeq(Declaration d, DeclarationList dl) {
    head=d;
    rest=dl;
}
public TypeEnvironment extract() throws DeclException {
    TypeEnvironment env = rest.extract();
    String head_var = head.getVariable();
        if (env.containsKey(head_var)) {
            throw new DeclException(head_var);
        } else {
            env.put(head_var,head.getType());
            return env;
        }
}
```


Implémentation : DeclarationList.java III

```
}
```


Typage de listes instructions

- Jugement $\Gamma \vdash il$: il est correctement typée dans Γ.
- Règles :

$$\frac{}{\Gamma \vdash Nil}$$

$$\frac{\Gamma \vdash i \qquad \Gamma \vdash il}{\Gamma \vdash Seq(i, il)}$$

Implémentation : InstructionList.java I

```
abstract class InstructionList {
    abstract void checkTypes(TypeEnvironment env) throws TypeExcep
}

class Nil extends InstructionList {
    public Nil() {
    }
    public void checkTypes(TypeEnvironment env) {
    }
}

class Seq extends InstructionList {
    private Instruction head;
    private InstructionList rest;
```

Implémentation : `InstructionList.java II`

```
    public Seq(Instruction i, InstructionList il) {
        head=i;
        rest=il;
    }
    public void checkTypes(TypeEnvironment env) throws TypeException {
        head.checkTypes(env);
        rest.checkTypes(env);
    }
}
```


Typage d'instructions

- Jugement $\Gamma \vdash i$: i est correctement typée dans Γ.
- Règle pour l'affichage :

$$\frac{\Gamma \vdash e : int}{\Gamma \vdash Print(e)}$$

- Règle pour l'affectation :

$$\frac{\Gamma \vdash e : \Gamma(s)}{\Gamma \vdash Affect(s, e)}$$

Typage d'instructions

- Règle pour la conditionnelle :

$$\frac{\Gamma \vdash e : bool \qquad \Gamma \vdash il_1 \qquad \Gamma \vdash il_2}{\Gamma \vdash Cond(e, il_1, il_2)}$$

- Règle pour le boucle :

$$\frac{\Gamma \vdash e : bool \qquad \Gamma \vdash il}{\Gamma \vdash While(e, il)}$$

Implémentation : `Instruction.java I`

```
abstract class Instruction {
    abstract void checkTypes(TypeEnvironment env) throws TypeExcep
}

class Assignment extends Instruction {
    private Expression expression;
    private String variable;
    public Assignment(String v, Expression e) {
        expression=e;
        variable=v;
    }
    public void checkTypes(TypeEnvironment env) throws TypeExcepti
        expression.setType(env);
        if (expression.getType() != env.get(variable)) {
```

Implémentation : Instruction.java II

```
                throw new TypeException("Assignment to " + variable);
            }
        }
    }

    class Print extends Instruction {
        private Expression expression;
        public Print(Expression e) {
            expression=e;
        }
        public void checkTypes(TypeEnvironment env) throws TypeException {
            expression.setType(env);
            if (expression.getType() != Type.Int) {
                throw new TypeException("Print");
            }
```


Implémentation : Instruction.java III

```
        }
    }

    class While extends Instruction {
        private Expression condition;
        private InstructionList body;
        public While(Expression e, InstructionList il) {
            condition=e;
            body=il;
        }
        public void checkTypes(TypeEnvironment env) throws TypeExcepti
            condition.setType(env);
            if (condition.getType() != Type.Bool) {
                throw new TypeException("While");
            } else {
```


Implémentation : Instruction.java IV

```
                body.checkTypes(env);
            }
        }
    }

    class Conditional extends Instruction {
        private Expression condition;
        private InstructionList positiv;
        private InstructionList negativ;
        public Conditional(Expression e, InstructionList il1, InstructionLis
            condition=e;
            positiv=il1;
            negativ=il2;
        }
        public void checkTypes(TypeEnvironment env) throws TypeException {
```


Implémentation : Instruction.java V

```
            condition.setType(env);
            if (condition.getType() != Type.Bool) {
                throw new TypeException("Conditional");
            } else {
                positiv.checkTypes(env);
                negativ.checkTypes(env);
            }
        }
    }
```

Deux approches

- Les valeurs obtenues de l'évaluation d'une expression peuvent être de type différent (ici : *bool* ou *int*).
- Première solution : Construire un type *somme* qui réunit toutes les valeurs de tous les types.
- Deuxième solution : Des environnements de valeurs et des méthodes d'évaluation différents pour les types différents.
- La deuxième solution est plus simple tant qu'on n'a que deux types, la première solution est plus générique.

Approche 2 : utiliser deux environnements

- Deux environnements :

$$\begin{aligned}\Gamma_i &: \Sigma^* \to \mathbb{N} \\ \Gamma_b &: \Sigma^* \to \{int, bool\}\end{aligned}$$

- Les deux fonctions sont partielles.
- Le typage nous garantie que

$$\text{domaine}(\Gamma_i) \cap \text{domaine}(\Gamma_b) = \emptyset$$

Jugements pour l'évaluation et l'exécution

- Forme des jugements pour l'évaluation :

$$\Gamma_i, \Gamma_b \vdash e \Rightarrow_i v_i$$
$$\Gamma_i, \Gamma_b \vdash e \Rightarrow_b v_b$$

où $v_i \in \mathbb{N}$, $v_b \in \{true, false\}$.

- Jugements et règles pour le traitement des déclarations : omis.
- Forme des jugements pour les instructions et les listes d'instructions :

$$\Gamma_i, \Gamma_b \vdash i \Rightarrow \Gamma'_i, \Gamma'_b$$
$$\Gamma_i, \Gamma_b \vdash il \Rightarrow \Gamma'_i, \Gamma'_b$$

Règles : transparents suivants.

Règles pour l'évaluation des expressions

$$\frac{}{\Gamma_i, \Gamma_b \vdash Int(n) \to_i n} \quad n \in \mathbb{N}$$

$$\frac{}{\Gamma_i, \Gamma_b \vdash True \to_b true} \qquad \frac{}{\Gamma_i, \Gamma_b \vdash False \to_b false}$$

$$\frac{}{\Gamma_i, \Gamma_b \vdash Ident(s) \to_i \Gamma_i(s)} \quad s \in \Sigma^*, \text{si } s \in \text{domaine}(\Gamma_i)$$

$$\frac{}{\Gamma_i, \Gamma_b \vdash Ident(s) \to_b \Gamma_b(s)} \quad s \in \Sigma^*, \text{si } s \in \text{domaine}(\Gamma_b)$$

Règles pour l'évaluation des expressions

$$\frac{\Gamma_i, \Gamma_b \vdash e_1 \rightarrow_i v_1 \qquad \Gamma_i, \Gamma_b \vdash e_2 \rightarrow_i v_2}{\Gamma_i, \Gamma_b \vdash \mathit{Plus}(e_1, e_2) \rightarrow_i v_1 +_{\mathit{int}} v_2}$$

$$\frac{\Gamma_i, \Gamma_b \vdash e_1 \rightarrow_i v_1 \qquad \Gamma_i, \Gamma_b \vdash e_2 \rightarrow_i v_2}{\Gamma_i, \Gamma_b \vdash \mathit{Equal}(e_1, e_2) \rightarrow_b \mathit{true}} \quad \text{si } v_1 = v_2$$

$$\frac{\Gamma_i, \Gamma_b \vdash e_1 \rightarrow_i v_1 \qquad \Gamma_i, \Gamma_b \vdash e_2 \rightarrow_i v_2}{\Gamma_i, \Gamma_b \vdash \mathit{Equal}(e_1, e_2) \rightarrow_b \mathit{false}} \quad \text{si } v_1 \neq v_2$$

et pareil pour les autres opérateurs.

Implémentation : Expression.java (début) I

```
abstract class Expression {
    Type type;
    final public Type getType() {
        return this.type;
    }
    abstract void setType(TypeEnvironment env) throws TypeExceptio
    // default, will be redefined for integer valued expressions
    public int evalInt(IntEnvironment ienv, BoolEnvironment benv)
        throws EvalException {
        throw new EvalException();
    }
    // default, will be redefined for bool valued expressions
    public boolean evalBool(IntEnvironment ienv, BoolEnvironment b
        throws EvalException {
```


Implémentation : Expression.java (début) II

```
        throw new EvalException();
    }
}

class Int extends Expression {
    private int value;
    public Int (int i) {
        value=i;
    }
    public void setType(TypeEnvironment env) {
        this.type=Type.Int;
    }
    public int evalInt(IntEnvironment ienv, BoolEnvironment benv) {
        return value;
    }
```


Implémentation : Expression.java (début) III

```
}

class True extends Expression {
    public True () {
    }
    public void setType(TypeEnvironment env) {
        this.type=Type.Bool;
    }
    public boolean evalBool(IntEnvironment ienv, BoolEnvironment b
        return true;
    }
}

class Ident extends Expression {
    private String name;
```

Implémentation : Expression.java (début) IV

```
    public Ident (String s) {
        name=s;
    }
    public void setType(TypeEnvironment env) {
        this.type=env.get(name);
    }
    public int evalInt(IntEnvironment ienv, BoolEnvironment benv) {
        return ienv.get(name);
    }
    public boolean evalBool(IntEnvironment ienv, BoolEnvironment benv) {
        return benv.get(name);
    }
}

class Sum extends Expression {
```


Implémentation : Expression.java (début) V

```
    private Expression left;
    private Expression right;
    public Sum(Expression e1, Expression e2) {
        left=e1;
        right=e2;
    }
    public void setType(TypeEnvironment env) throws TypeException{
        left.setType(env);
        right.setType(env);
        if (left.getType()==Type.Int
            && right.getType()==Type.Int) {
            this.type=Type.Int;
        } else {
            throw new TypeException("Sum");
        }
```


Implémentation : Expression.java (début) VI

```
    }
    public int evalInt(IntEnvironment ienv, BoolEnvironment benv)
        throws EvalException {
        int value_left = left.evalInt(ienv,benv);
        int value_right = right.evalInt(ienv,benv);
        return value_left + value_right;
    }
}

class Equal extends Expression {
    private Expression left;
    private Expression right;
    public Equal(Expression e1, Expression e2) {
        left=e1;
        right=e2;
```


Implémentation : Expression.java (début) VII

```
    }
    public void setType(TypeEnvironment env) throws TypeException{
        left.setType(env);
        right.setType(env);
        if (left.getType()==Type.Int
            && right.getType()==Type.Int) {
            this.type=Type.Bool;
        } else {
            throw new TypeException("Equal");
        }
    }
    public boolean evalBool(IntEnvironment ienv, BoolEnvironment b
        throws EvalException {
        int value_left = left.evalInt(ienv,benv);
        int value_right = right.evalInt(ienv,benv);
```

Implémentation : Expression.java (début) VIII

```
        return value_left == value_right;
    }
}
```


Règles d'exécution de listes d'instructions

$$\frac{}{\Gamma_i, \Gamma_b \vdash Nil \Rightarrow \Gamma_i, \Gamma_b}$$

$$\frac{\Gamma_i, \Gamma_b \vdash i \Rightarrow \Gamma'_i, \Gamma'_b \qquad \Gamma'_i, \Gamma'_b \vdash il \Rightarrow \Gamma''_i, \Gamma''_b}{\Gamma_i, \Gamma_b \vdash Seq(i, il) \Rightarrow \Gamma''_i, \Gamma''_b}$$

Implémentation : InstructionList.java I

```
abstract class InstructionList {
    abstract void checkTypes(TypeEnvironment env) throws TypeException;
    abstract void exec(IntEnvironment ienv, BoolEnvironment benv)
        throws EvalException;
}

class Nil extends InstructionList {
    public Nil() {
    }
    public void checkTypes(TypeEnvironment env) {
    }
    public void exec(IntEnvironment ienv, BoolEnvironment benv) {
    }
}
```


Implémentation : InstructionList.java II

```
class Seq extends InstructionList {
    private Instruction head;
    private InstructionList rest;
    public Seq(Instruction i, InstructionList il) {
        head=i;
        rest=il;
    }
    public void checkTypes(TypeEnvironment env) throws TypeExcepti
        head.checkTypes(env);
        rest.checkTypes(env);
    }
    public void exec(IntEnvironment ienv, BoolEnvironment benv)
        throws EvalException {
        head.exec(ienv,benv);
```

Implémentation : `InstructionList.java` III

```
            rest.exec(ienv, benv);
        }
    }
```


Règles pour l'exécution des instructions

- Règle pour l'affichage :

$$\frac{}{\Gamma_i, \Gamma_b \vdash \mathit{Print}(e) \Rightarrow \Gamma_i, \Gamma_b}$$

- Règles pour l'affectation :

$$\frac{\Gamma_i, \Gamma_b \vdash e \Rightarrow_i v}{\Gamma_i, \Gamma_b \vdash \mathit{Affect}(s, e) \Rightarrow \Gamma_i[s \mapsto v], \Gamma_b}$$

$$\frac{\Gamma_i, \Gamma_b \vdash e \Rightarrow_b v}{\Gamma_i, \Gamma_b \vdash \mathit{Affect}(s, e) \Rightarrow \Gamma_i, \Gamma_b[s \mapsto v]}$$

Règles pour l'exécution des instructions

- Première règle pour la conditionnelle :

$$\frac{\Gamma_i, \Gamma_b \vdash e \Rightarrow_b \mathit{true} \qquad \Gamma_i, \Gamma_b \vdash il_1 \Rightarrow \Gamma'_i, \Gamma'_b}{\Gamma_i, \Gamma_b \vdash \mathit{Cond}(e, il_1, il_2) \Rightarrow \Gamma'_i, \Gamma'_b}$$

- Deuxième règle pour la conditionnelle :

$$\frac{\Gamma_i, \Gamma_b \vdash e \Rightarrow_b \mathit{false} \qquad \Gamma_i, \Gamma_b \vdash il_2 \Rightarrow \Gamma'_i, \Gamma'_b}{\Gamma_i, \Gamma_b \vdash \mathit{Cond}(e, il_1, il_2) \Rightarrow \Gamma'_i, \Gamma'_b}$$

Règles pour l'exécution des instructions

Règles pour la boucle :

$$\frac{\Gamma_i, \Gamma_b \vdash e \Rightarrow_b \mathit{false}}{\Gamma_i, \Gamma_b \vdash \mathit{While}(e, il) \Rightarrow \Gamma_i, \Gamma_b}$$

$$\frac{\Gamma_i, \Gamma_b \vdash e \Rightarrow_b \mathit{true} \quad \Gamma_i, \Gamma_b \vdash il \Rightarrow \Gamma'_i, \Gamma'_b \quad \Gamma'_i, \Gamma'_b \vdash \mathit{While}(e, il) \Rightarrow \Gamma''_i, \Gamma''_b}{\Gamma_i, \Gamma_b \vdash \mathit{While}(e, il) \Rightarrow \Gamma''_i, \Gamma''_b}$$

Implémentation : Instruction.java I

```
abstract class Instruction {
    abstract void checkTypes(TypeEnvironment env) throws TypeException;
    abstract void exec(IntEnvironment ienv, BoolEnvironment benv)
        throws EvalException;
}

class Assignment extends Instruction {
    private Expression expression;
    private String variable;
    public Assignment(String v, Expression e) {
        expression=e;
        variable=v;
    }
    public void checkTypes(TypeEnvironment env) throws TypeException {
```


Implémentation : Instruction.java II

```
        expression.setType(env);
        if (expression.getType() != env.get(variable)) {
            throw new TypeException("Assignment␣to␣" + variable);
        }
    }
    public void exec(IntEnvironment ienv,BoolEnvironment benv)
        throws EvalException {
        if (expression.getType()==Type.Int) {
            ienv.put(variable,expression.evalInt(ienv,benv));
        } else {
            benv.put(variable,expression.evalBool(ienv,benv));
        }
    }
}
```


Implémentation : Instruction.java III

```
class Print extends Instruction {
    private Expression expression;
    public Print(Expression e) {
        expression=e;
    }
    public void checkTypes(TypeEnvironment env) throws TypeException {
        expression.setType(env);
        if (expression.getType() != Type.Int) {
            throw new TypeException("Print");
        }
    }
    public void exec(IntEnvironment ienv,BoolEnvironment benv)
        throws EvalException {
        System.out.println(expression.evalInt(ienv,benv));
    }
```


Implémentation : Instruction.java IV

```
}

class Conditional extends Instruction {
    private Expression condition;
    private InstructionList positiv;
    private InstructionList negativ;
    public Conditional(Expression e, InstructionList il1, Instruct
        condition=e;
        positiv=il1;
        negativ=il2;
    }
    public void checkTypes(TypeEnvironment env) throws TypeExcepti
        condition.setType(env);
        if (condition.getType() != Type.Bool) {
            throw new TypeException("Conditional");
```

Implémentation : Instruction.java V

```
            } else {
                positiv.checkTypes(env);
                negativ.checkTypes(env);
            }
        }
        public void exec(IntEnvironment ienv,BoolEnvironment benv)
            throws EvalException {
            if (condition.evalBool(ienv,benv)) {
                positiv.exec(ienv,benv);
            } else {
                negativ.exec(ienv,benv);
            }
        }
    }
```


Implémentation : Instruction.java VI

```
    class While extends Instruction {
        private Expression condition;
        private InstructionList body;
        public While(Expression e, InstructionList il) {
            condition=e;
            body=il;
        }
        public void checkTypes(TypeEnvironment env) throws TypeExcepti
            condition.setType(env);
            if (condition.getType() != Type.Bool) {
                throw new TypeException("While");
            } else {
                body.checkTypes(env);
            }
        }
```


Implémentation : Instruction.java VII

```
        public void exec(IntEnvironment ienv,BoolEnvironment benv)
            throws EvalException {
            if (condition.evalBool(ienv,benv)) {
                body.exec(ienv,benv);
                this.exec(ienv,benv);
            }
        }
    }
```