Inférence des types

Présenté par :

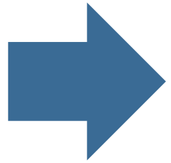
- Buisson Kilian
- Therin Ivann
- Chonger Thomas

SOMMAIRE

Répartition du travail

PrimitiveType

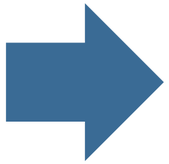
UnknownType



CHONGER Thomas

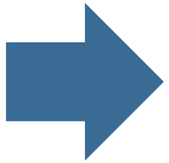
ArrayType

FunctionType



BUISSON Kilian

TypeVisitor



THERIN Ivann

PrimitiveType

```java
@Override
public Map<UnknownType, Type> unify(Type t) {
    // Rempli
    if (t instanceof PrimitiveType) {
        PrimitiveType other = (PrimitiveType) t;
        if (this.type == other.type) {
            return new HashMap<>();
        } else {
            return null;
        }
    } else if (t instanceof UnknownType) {
        // CLÉ : Un PrimitiveType (this) s'unifie
        return t.unify(this);
    }
    return null;
}
```

Unification

```java
@Override
public Type substitute(UnknownType v, Type t) {
    // Rempli
    return this;
}
```

Substitution

UnkownType

```java
@Override
public Map<UnknownType, Type> unify(Type t) {
    // Rempli
    if (t.equals(this)) {
        return new HashMap<>();
    }

    // ÉTAPE CRITIQUE : Test d'occurrence (pour éviter alpha -> Array<alpha>)
    if (t.contains(this)) {
        // Si 't' contient la variable de type 'this', échec d'unification.
        return null;
    }

    // Création de la substitution : {this -> t}
    Map<UnknownType, Type> subs = new HashMap<>();
    subs.put(this, t);

    return subs;
}
```

Unification

```java
@Override
public Type substitute(UnknownType v, Type t) {
    // Rempli
    if (this.equals(v)) {
        return t; // C'est la variable de type, on la remplace.
    }
    return this; // Ce n'est pas la variable à remplacer.
}
```

Substitution

ArrayType

```java
@Override
public Map<UnknownType, Type> unify(Type t) {
    // Rempli
    if (t instanceof ArrayType) {
        // ...
        ArrayType other = (ArrayType) t;
        return this.tabType.unify(other.tabType);
    } else if (t instanceof UnknownType) {
        // Délègue à UnknownType, et RETOURNE le résultat
        return t.unify(this);
    }
    return null; // Échec
}
```

Unification

```java
@Override
public Type substitute(UnknownType v, Type t) {
    // Rempli
    if (this.equals(t)) return t;


    Type newTabType = this.tabType.substitute(v, t);

    if (newTabType == this.tabType) {
        return this;
    }
    return new ArrayType(newTabType);
}
```

Substitution

FunctionType

```java
@Override
public Map<UnknownType, Type> unify(Type t) {
    // Rempli
    if (!(t instanceof FunctionType)) return null;

    FunctionType other = (FunctionType) t;
    if (this.argsTypes.size() != other.argsTypes.size()) return null;

    Map<UnknownType, Type> subs = new HashMap<>();

    // 1. Unification du type de retour
    Map<UnknownType, Type> subReturn = this.returnType.unify(other.returnType);
    if (subReturn == null) return null;
    subs.putAll(subReturn); // Ajout des substitutions trouvées

    // 2. Unification séquentielle des arguments (avec composition)
    for (int i = 0; i < argsTypes.size(); i++) {

        // ÉTAPE CRITIQUE : Appliquer toutes les substitutions courantes (subs) aux types AVANT l'unification!
        Type t1 = this.argsTypes.get(i).substituteAll(subs);
        Type t2 = other.argsTypes.get(i).substituteAll(subs);

        Map<UnknownType, Type> subArg = t1.unify(t2);
        if (subArg == null) return null;

        // COMPOSITION DES SUBSTITUTIONS
        // Créer un nouvel ensemble de substitutions en composant subs avec subArg.

        Map<UnknownType, Type> composedSubs = new HashMap<>();

        // Appliquer les nouvelles substitutions (subArg) aux valeurs des substitutions existantes (subs)
        for (Map.Entry<UnknownType, Type> entry : subs.entrySet()) {
            composedSubs.put(entry.getKey(), entry.getValue().substituteAll(subArg));
        }

        // Ajouter les nouvelles substitutions (subArg), elles priment sur les anciennes
        composedSubs.putAll(subArg);

        subs = composedSubs; // Mise à jour de la map globale
    }
    return subs;
```

Unification

```java
@Override
public Type substitute(UnknownType v, Type t) {
    // Rempli
    // 1. Substitution du type de retour (récursif)
    Type newReturnType = this.returnType.substitute(v, t);

    // 2. Substitution des types d'arguments (récursif)
    ArrayList<Type> newArgsTypes = new ArrayList<>();
    boolean changed = false; // Flag pour optimiser

    for (Type argType : argsTypes) {
        Type newArgType = argType.substitute(v, t);
        newArgsTypes.add(newArgType);

        if (newArgType != argType) {
            changed = true;
        }
    }

    // Si des changements ont eu lieu dans le type de retour OU les arguments
    if (newReturnType != this.returnType || changed) {
        // Retourne une nouvelle instance de FunctionType avec les types mis à jour.
        return new FunctionType(newReturnType, newArgsTypes);
    }

    // Sinon, retourne l'instance actuelle (immuabilité)
    return this;
}
```

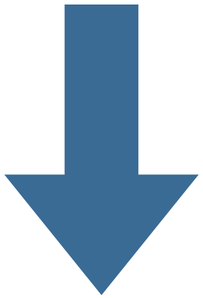
Substitution

TyperVisitor

Objectifs :

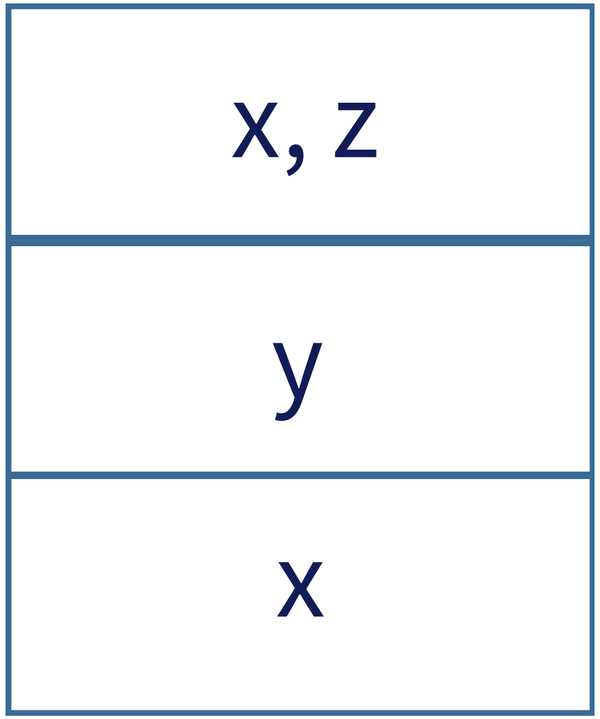
- Vérification
- Inférence

Map<UnknownType,Type>



Code → AST → TyperVisitor → Table des types

FunctionType



Pile

```
int main() {
    int x = 10;
    if (x > 0) {
        int y = 20;
        if (y == 20) {
            int x = 999;
            int z = y + 30;
            print(x);
        }
        print(x);
    }
    return 0;
}
```

Exemple de code

Point technique

```java
@Override  1 usage
public Type visitCall(grammarTCLParser.CallContext ctx) {
    String fctName = ctx.VAR().getText();
    Type tFct = lookup(fctName);

    if (!(tFct instanceof FunctionType)) {
        throw new RuntimeException("Type Error: L'identifiant " + fctName + " n'est pas une fonction.");
    }

    FunctionType signature = (FunctionType) instantiateType(tFct, new HashMap<UnknownType,UnknownType>());

    List<grammarTCLParser.ExprContext> callArgs = ctx.expr();

    if (signature.getNbArgs() != callArgs.size()) {
        throw new RuntimeException("Type Error: Nombre d'arguments incorrect pour " + fctName);
    }

    for (int i = 0; i < callArgs.size(); i++) {
        Type expectedArgType = signature.getArgsType(i);
        Type actualArgType = visit(callArgs.get(i));

        unifyAndApply(expectedArgType, actualArgType, ctx);
    }

    UnknownType tResult = new UnknownType();
    unifyAndApply(signature.getReturnType(), tResult, ctx);

    return tResult.substituteAll(this.types);
}
```

Exemple de code TCL :

```
auto test(auto x) {
    return x;
}

int main() {
    int val1 = test(10);
    bool val2 = test(true);
    return 0;
}
```

```
int calculerCarre(int n) {
    return n * n;
}

// Fonction qui vérifie si un nombre est pair
bool estPair(int n) {
    int reste = n % 2;
    return reste == 0;
}

int main() {
    int x = 4;
    int resultat = calculerCarre(x);
    bool check = estPair(resultat);

    if (check) {
        print(resultat);
    } else {
        int zero = 0;
        print(zero);
    }

    return 0;
}
```

```
--- Test du fichier : test_ok.tcl ---
Typage réussi !
Types résolus (Substitutions) :
{alpha0=INT, alpha1=BOOL}
-------------------------------------
```

Unifie (alpha0, int)

Unifie (alpha1, bool)

alpha0 := int

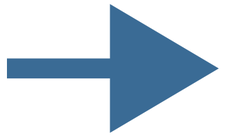
alpha1 := bool

```
// Fonction retournant un en
int calculer(int a) {
    int res = a * 2;
    return res;
}

int main() {
    // 'val' est déclaré en '
    auto val = calculer(10);

    print(val);
    return 0;
}
```

```
--- Test du fichier : test_ok.tcl ---
Typage réussi !
Types résolus (Substitutions) :
{alpha0=INT, alpha1=INT}
-------------------------------------
```

Unifie (alpha0, int)

Unifie (alpha1, alpha0)



alpha0 := int

alpha1 := int

```
--- Test du fichier : test_ok.tcl ---
Typage réussi !
Types résolus (Substitutions) :
{alpha0=INT}
----------------------------------
```

```c
int main() {
    int[] notes = {12, 15, 18};
    int total = 0;
    int i = 0;

    for (i = 0 , i < 3 , i = i + 1 ) {
        int n = notes[i];
        total = total + n;
    }

    print(total);
    return 0;
}
```

Unifie (alpha0, int)

alpha0 := int

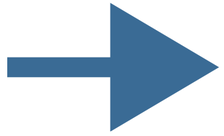
```
--- Test du fichier : test_ok.tcl ---
Typage échoué (INATTENDU) !
------------------------------------
Erreur de type : Type Error à la ligne 11 : Impossible d'unifier INT et BOOL.
```

```
bool estValide(int n) {
    return n > 0;
}

int main() {
    int x = 5;
    // ERREUR : Tentative d'assi
    int resultat = estValide(x);

    return 0;
}
```

estValide(n) → alpha0 := bool

resultat → alpha1 := int

Unifie (alpha0, alpha1)



Erreur

```
--- Test du fichier : test_ok.tcl ---
Typage réussi !
Types résolus (Substitutions) :
{alpha2=INT[], alpha3=INT, alpha4=INT[], alpha5=BOOL[], alpha6=BOOL, alpha7=BOOL[]}
```

```
auto f(auto x){
    return {x};
    }

int main(){
    int[] x = f(0);
    bool[] y = f(true);
    return 0;
}
```

Unifie (f(0),tab[0])

Unifie (x, f(0))

0 → int

f := int → tab[int]

x := tab[int]

En général :

F := A → tab[A]

```
auto f () {
    if (0 == 1) {
        return true;
    }
    else {
        return 0;
    }
    return {x};
}
```

```
--- Test du fichier : test_ok.tcl ---
Typage échoué (INATTENDU) !
-------------------------------------
```

Bool

Int

tab[int]

Types de retour différents

Erreur

Conclusion

Vérification du typage

Typage incorrect : Erreur → Donne les types non unifiables

Typage correct : Donne les types résolus (substitutions)

→ Inférence de type lorsque possible