Fondations pour Scala : sémantique et preuve des types virtuels

Défense publique de thèse

Vincent Cremet

EPFL

8 septembre 2006

Contexte de la thèse

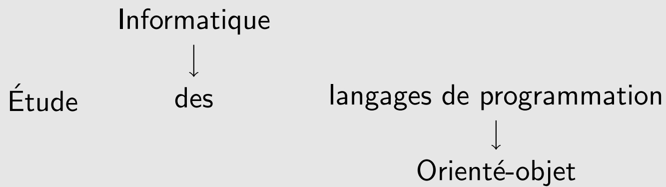
Informatique

Contexte de la thèse

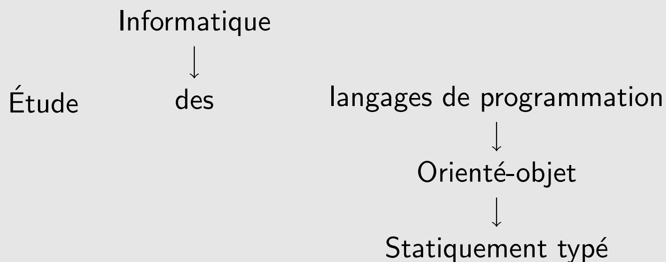
Informatique

Étude des langages de programmation

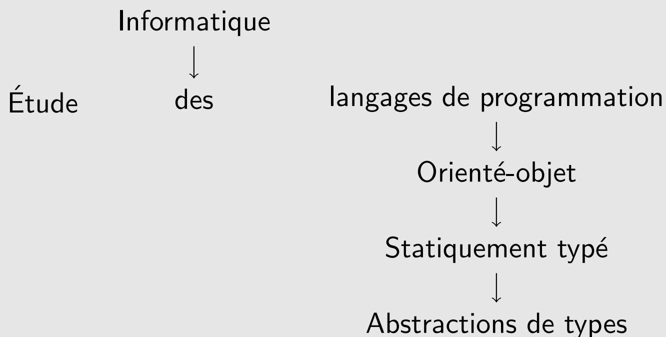
Contexte de la thèse



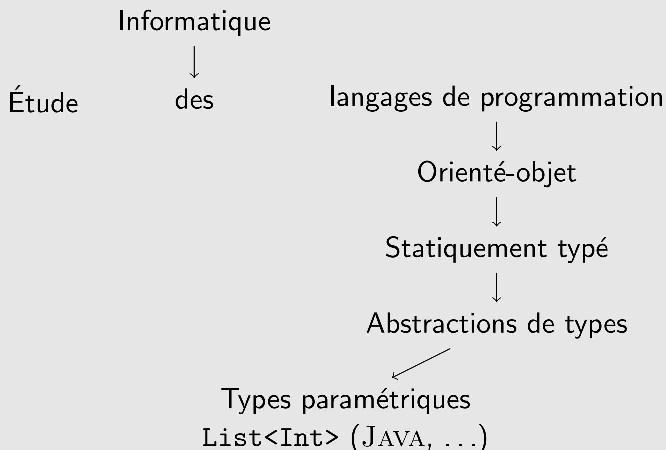
Contexte de la thèse



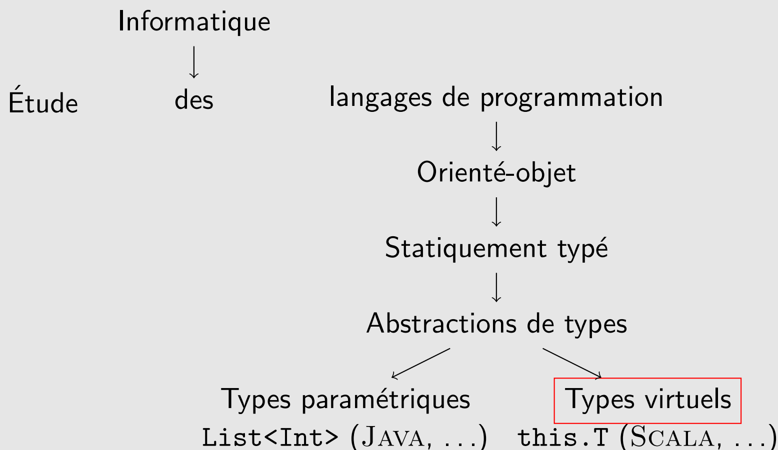
Contexte de la thèse



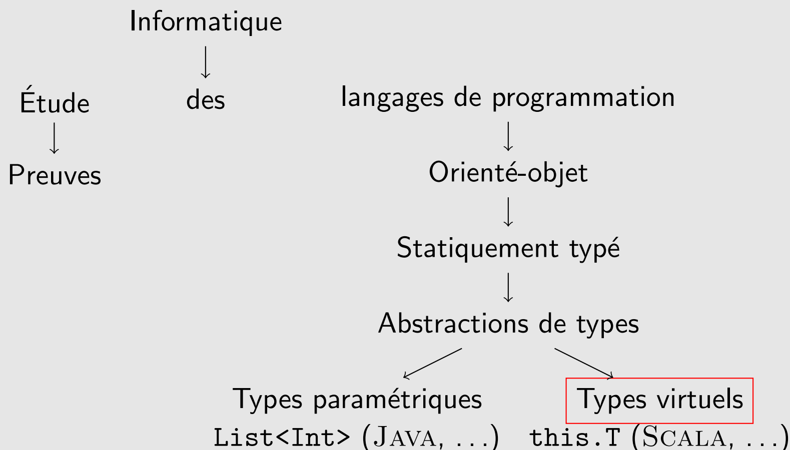
Contexte de la thèse



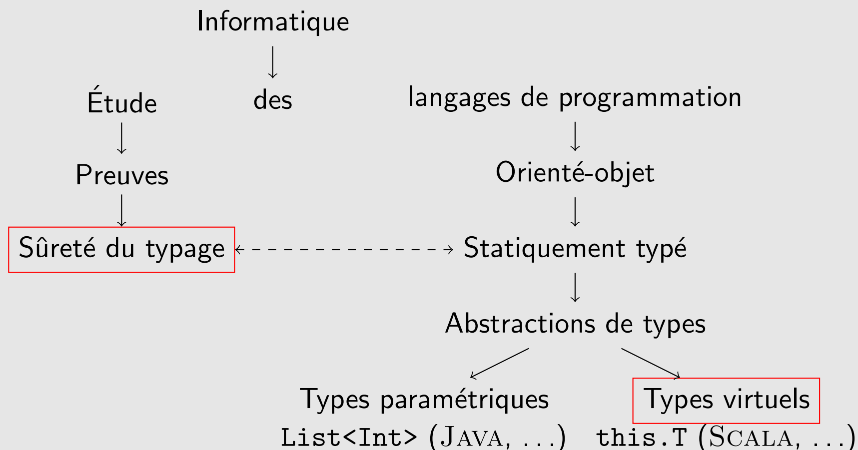
Contexte de la thèse



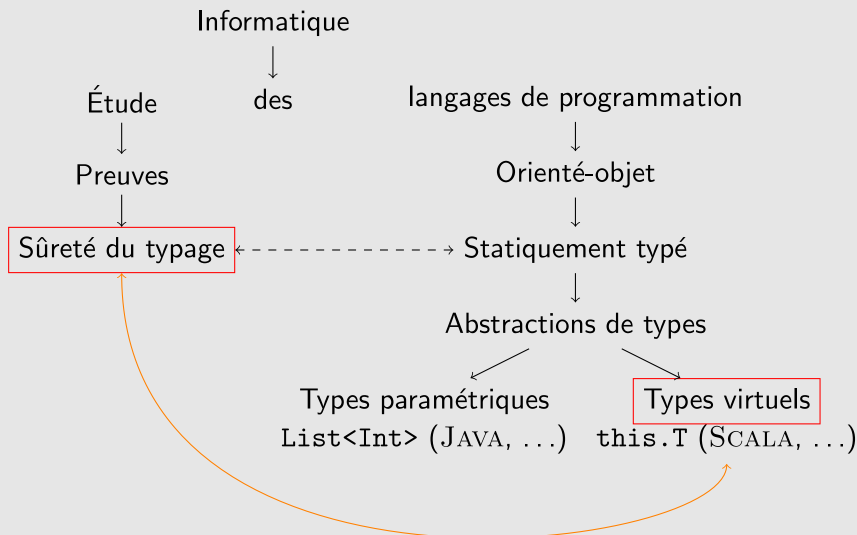
Contexte de la thèse



Contexte de la thèse



Contexte de la thèse



Plan

1. Preuve de correction des types virtuels
   - Généricité et types virtuels
   - Sûreté du typage
   - Vers un sous-typage sûr

Qu'est ce qu'un type virtuel ?

- Un champ d'une classe qui contient un type, plutôt qu'un objet.
- Sa valeur est exploitée à la compilation, plutôt qu'à l'exécution.

Exemple des animaux

```
abstract class Aliment

abstract class Animal {
  type Regime <: Aliment
  def manger(x: this.Regime)
}
```

```
class Herbe extends Aliment {
  def ruminer() = ...
}

class Vache extends Animal {
  type Regime = Herbe
  def manger(x: this.Regime) =
    x.ruminer()
}
```

Encodage des types paramétriques

Types paramétriques

```
class Aliment

class Animal[Regime <: Aliment] {
  def manger(x: Regime)
}


class Herbe extends Aliment {
  def ruminer() = ...
}

class Vache extends Animal[Herbe] {
  def manger(x: Herbe) =
    x.ruminer()
}
```

Types virtuels

```
class Aliment

class Animal {
  type Regime <: Aliment
  def manger(x: this.Regime)
}

class Herbe extends Aliment {
  def ruminer() = ...
}

class Vache extends Animal {
  type Regime = Herbe
  def manger(x: this.Regime) =
    x.ruminer()
}
```

Types virtuels et paramètres de type

Question : Peut-on encoder les types virtuels avec les paramètres de types ?

Types virtuels

```
class List {
  type X
  val head: this.X
  val tail: List
}
```

On a une infinité de types inconnus.

```
l: List
l.head           : l.X
l.tail.head      : l.tail.X
l.tail.tail.head: l.tail.tail.X
...
```

Paramètres de type

```
class List[X] {
  val head: X
  val tail: List[?]
}
```

La traduction nécessite le type joker ? de JAVA.

Types virtuels et paramètres de type (suite)

Types virtuels

```
class List {
  type X
  val head       : this.X
  val tail       : List
  val elemOfTail : this.tail.X
}
```

Types paramétriques

```
class List[X,Y] {
  val head       : X
  val tail       : List[Y,?]
  val elemOfTail : Y
}
```

- Pour un même symbol de type `X`, un paramètre de type par occurrence différente :
  `X` pour `this.X`, et
  `Y` pour `this.tail.X`.
- Il n'est pas clair qu'on puisse toujours faire la traduction.
- En tout cas, la traduction n'est pas compositionnelle : elle nécessite une analyse globale du programme.

Sûreté du typage pour les types virtuels

- Les types virtuels sont plus généraux que les types paramétriques.
- Les types virtuels sont le mécanisme de base pour l'abstraction de type en SCALA.
- Plusieurs calculs existants avec une utilisation très générale des types virtuels (ν-OBJ, etc). Mais, pas de preuve complètement formelle.

Sûreté du typage pour les types virtuels

- Les types virtuels sont plus généraux que les types paramétriques.
- Les types virtuels sont le mécanisme de base pour l'abstraction de type en SCALA.
- Plusieurs calculs existants avec une utilisation très générale des types virtuels (ν-OBJ, etc). Mais, pas de preuve complètement formelle.
- On veut une preuve complètement formelle qui puisse être vérifiée par un ordinateur (COQ)
- En contrepartie : On se limite à un calcul moins expressif.

Langage d'étude

On considère une extension de Featherweight Java (FJ). Deux nouvelles sortes de déclarations dans les classes pour :

1. Déclarer un nouveau type : `type L >: T <: U`.
   `T` et `U` sont respectivement les bornes inférieures et supérieures du type virtuel `L`.
2. Instancier un type existant : `type L = T`.

Un type est soit

1. un type classe : `C`, ou
2. un type virtuel : `p.L`.

Limitation : Le préfixe `p` d'un type virtuel est forcément une expression simple, c-à-d : `this`, une variable locale ou le paramètre d'une fonction (mais pas une suite de sélections comme en SCALA).

Sûreté du typage

Définition :

Sûreté du typage = "tous les programmes bien typés sont sûrs"

- Un programme est dit **bien typé** si les types des sous-expressions du programme sont cohérents entre eux [*propriété statique*].

  Par ex. : `3 * "coucou"` est mal typé.
- Un programme est dit **sûr** s'il s'exécute correctement (pas de champ ou méthode non trouvés) [*propriété dynamique*].

Structure générale de la preuve

- Une sémantique opérationnelle à petits pas (évaluation = réduction itérée).

Théorème (Sûreté du typage)

Un terme bien typé dans l'environnement vide, soit diverge, soit se réduit en une valeur.

- Décomposition classique de la preuve en :
  - un lemme de progrès : un terme bien typé qui n'est pas une valeur est réductible.
  - un lemme de préservation du typage (*subject-reduction* en anglais) : la propriété d'être bien typé est préservée par réduction.

Sous-typage et sous-classage

Première configuration non sûre

`A` sous-type de `B`, mais **pas** `A` sous-classe de `B`.

Preuve :

```
class A { }
class B {
  def m(): B = new B()
}
new A().m()
```

- Si `A` est un sous-type de `B`, alors le programme est accepté à la compilation.
- Comme `A` n'est pas une sous-classe de `B`, la méthode `m` n'est pas trouvée à l'exécution.

Sous-typage et sous-classage (suite)

Deuxième configuration non sûre

`A` sous-type de `T`, `T` sous-type de `B`, mais **pas** `A` sous-classe de `B`.

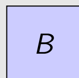
Preuve :

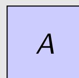
```
def id1(x: A): T = x
def id2(x: T): B = x
def id (x: A): B = id2(id1(x))

id(new A()).m()
```

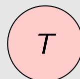
- On simule la transitivité en composant des fonctions "identité".
- Se généralise à un nombre quelconque d'étapes de sous-typage.
- Configurations dangereuses : permet de détecter rapidement des programmes non sûrs.

Graphe des symboles

```
class A {
  type T <: A          // (1)
}
class B extends A {    // (2)
  type T = B           // (3)
}
```



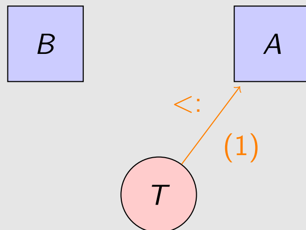




- Noeuds : les symboles de type.
- Arêtes : déclarations de types, héritage, affectation de type.

Graphe des symboles

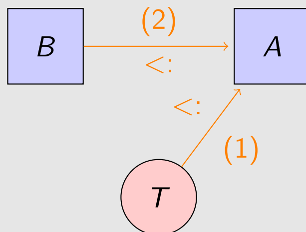
```
class A {
  type T <: A          // (1)
}
class B extends A {    // (2)
  type T = B           // (3)
}
```



- Noeuds : les symboles de type.
- Arêtes : déclarations de types, héritage, affectation de type.

Graphe des symboles

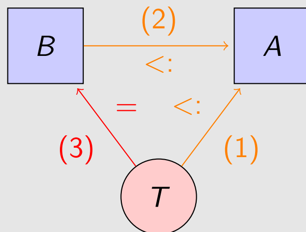
```
class A {
  type T <: A           // (1)
}
class B extends A {     // (2)
  type T = B            // (3)
}
```



- Noeuds : les symboles de type.
- Arêtes : déclarations de types, héritage, affectation de type.

Graphe des symboles

```
class A {
  type T <: A          // (1)
}
class B extends A {    // (2)
  type T = B           // (3)
}
```



- Noeuds : les symboles de type.
- Arêtes : déclarations de types, héritage, affectation de type.

Règles de sous-typage naïves

- Le sous-typage inclut le sous-classage.

$$(\textsc{S-Extends})\ \frac{\texttt{class}\ C\ \texttt{extends}\ C'\ \{\ \cdots\ \}}{\Gamma \vdash C <: C'}$$

Règles de sous-typage naïves

- Le sous-typage inclut le sous-classage.

$$(\textsc{S-Extends})\ \frac{\texttt{class}\ C\ \texttt{extends}\ C'\ \{\ \cdots\ \}}{\Gamma\ \vdash\ C <: C'}$$

- Un type virtuel est sous-type de sa borne supérieure.

$$(\textsc{S-Up})\ \frac{\Gamma\ \vdash\ p : C \qquad (\texttt{type}\ L <: T) \in C}{\Gamma\ \vdash\ p.L <: T[\texttt{this}\backslash p]}$$

Règles de sous-typage naïves

- Le sous-typage inclut le sous-classage.

$$(\textsc{S-Extends})\ \frac{\texttt{class}\ C\ \texttt{extends}\ C'\ \{\ \cdots\ \}}{\Gamma\ \vdash\ C <: C'}$$

- Un type virtuel est sous-type de sa borne supérieure.

$$(\textsc{S-Up})\ \frac{\Gamma\ \vdash\ p : C \qquad (\texttt{type}\ L <: T) \in C}{\Gamma\ \vdash\ p.L <: T[\texttt{this}\backslash p]}$$

- Un type virtuel est super-type de sa valeur.

$$(\textsc{S-Right})\ \frac{\Gamma\ \vdash\ p : C \qquad (\texttt{type}\ L = T) \in C}{\Gamma\ \vdash\ T[\texttt{this}\backslash p] <: p.L}$$

Règles de sous-typage naïves

- Le sous-typage inclut le sous-classage.

$$(\text{S-Extends})\ \frac{\texttt{class}\ C\ \texttt{extends}\ C'\ \{\ \cdots\ \}}{\Gamma \vdash C <: C'}$$

- Un type virtuel est sous-type de sa borne supérieure.

$$(\text{S-Up})\ \frac{\Gamma \vdash p : C \qquad (\texttt{type}\ L <: T) \in C}{\Gamma \vdash p.L <: T[\texttt{this}\backslash p]}$$

- Un type virtuel est super-type de sa valeur.

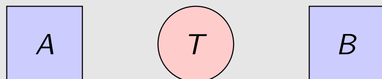
$$(\text{S-Right})\ \frac{\Gamma \vdash p : C \qquad (\texttt{type}\ L = T) \in C}{\Gamma \vdash T[\texttt{this}\backslash p] <: p.L}$$

- Le sous-typage est transitif.

$$(\text{S-Trans})\ \frac{\Gamma \vdash T <: S \qquad \Gamma \vdash S <: U}{\Gamma \vdash T <: U}$$

Les règles naïves ne sont pas sûres

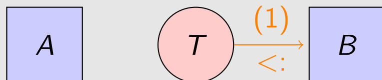
```
class A { }
class B { }
class C {
  type T <: B      // 1
  type T = A       // 2
}
```



- Problème : programme bien formé et A sous-type de B (config. 1).
- Solution : interdire de suivre les flèches $\xrightarrow{=}$ à contre-courant (interdire S-Right) ?

Les règles naïves ne sont pas sûres

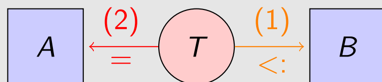
```
class A { }
class B { }
class C {
  type T <: B      // 1
  type T = A       // 2
}
```



- Problème : programme bien formé et A sous-type de B (config. 1).
- Solution : interdire de suivre les flèches $\xrightarrow{=}$ à contre-courant (interdire S-Right) ?

Les règles naïves ne sont pas sûres

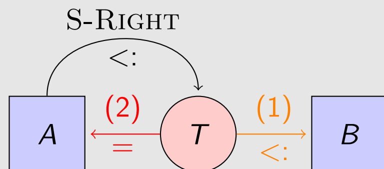
```
class A { }
class B { }
class C {
  type T <: B      // 1
  type T = A       // 2
}
```



- Problème : programme bien formé et A sous-type de B (config. 1).
- Solution : interdire de suivre les flèches $\xrightarrow{=}$ à contre-courant (interdire S-Right) ?

Les règles naïves ne sont pas sûres

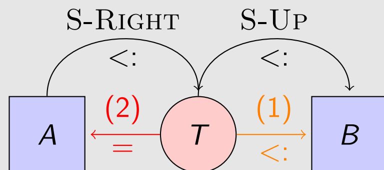
```
class A { }
class B { }
class C {
  type T <: B      // 1
  type T = A       // 2
}
```



- Problème : programme bien formé et A sous-type de B (config. 1).
- Solution : interdire de suivre les flèches $\xrightarrow{=}$ à contre-courant (interdire S-RIGHT) ?

Les règles naïves ne sont pas sûres

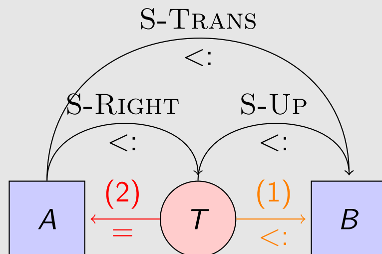
```
class A { }
class B { }
class C {
  type T <: B      // 1
  type T = A       // 2
}
```



- Problème : programme bien formé et A sous-type de B (config. 1).
- Solution : interdire de suivre les flèches $\xrightarrow{=}$ à contre-courant (interdire S-Right) ?

Les règles naïves ne sont pas sûres

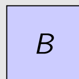
```
class A { }
class B { }
class C {
  type T <: B      // 1
  type T = A       // 2
}
```

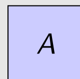


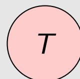
- Problème : programme bien formé et A sous-type de B (config. 1).
- Solution : interdire de suivre les flèches $\xrightarrow{=}$ à contre-courant (interdire S-Right) ?

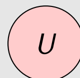
Nécessité d'aller à contre-courant

```
class A {
  type T <: A           // (1)
  type U <: this.T      // (2)
}
class B extends A {     // (3)
  type T = A            // (4)
  type U = B            // (5)
}
```







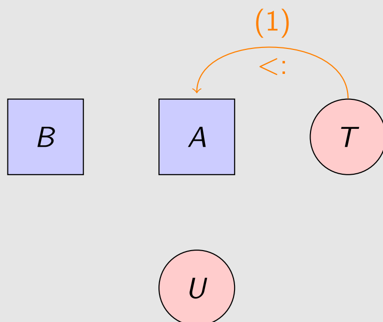


- Idée : On doit pouvoir converger vers un même type S :

- L'exemple précédent est bien rejeté.

Nécessité d'aller à contre-courant

```
class A {
  type T <: A          // (1)
  type U <: this.T     // (2)
}
class B extends A {    // (3)
  type T = A           // (4)
  type U = B           // (5)
}
```

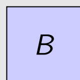


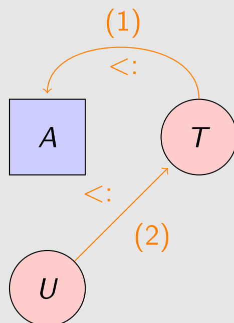
- Idée : On doit pouvoir converger vers un même type S :

- L'exemple précédent est bien rejeté.

Nécessité d'aller à contre-courant

```
class A {
  type T <: A          // (1)
  type U <: this.T     // (2)
}
class B extends A {    // (3)
  type T = A           // (4)
  type U = B           // (5)
}
```



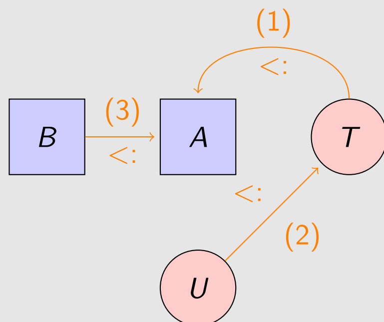


- Idée : On doit pouvoir converger vers un même type S :

- L'exemple précédent est bien rejeté.

Nécessité d'aller à contre-courant

```
class A {
  type T <: A            // (1)
  type U <: this.T       // (2)
}
class B extends A {      // (3)
  type T = A             // (4)
  type U = B             // (5)
}
```

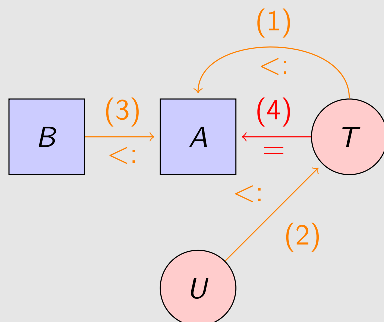


- Idée : On doit pouvoir converger vers un même type S :

- L'exemple précédent est bien rejeté.

Nécessité d'aller à contre-courant

```
class A {
  type T <: A          // (1)
  type U <: this.T     // (2)
}
class B extends A {    // (3)
  type T = A           // (4)
  type U = B           // (5)
}
```

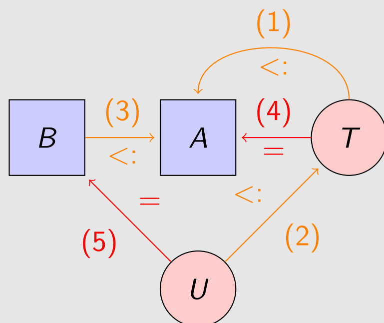


- Idée : On doit pouvoir converger vers un même type S :

- L'exemple précédent est bien rejeté.

Nécessité d'aller à contre-courant

```
class A {
  type T <: A          // (1)
  type U <: this.T     // (2)
}
class B extends A {    // (3)
  type T = A           // (4)
  type U = B           // (5)
}
```

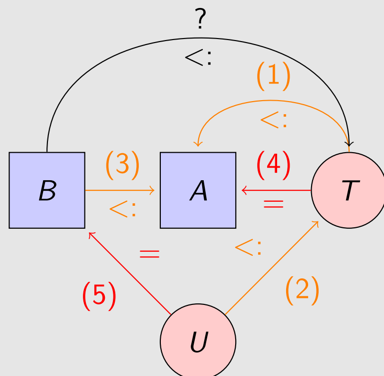


- Idée : On doit pouvoir converger vers un même type S :

- L'exemple précédent est bien rejeté.

Nécessité d'aller à contre-courant

```
class A {
  type T <: A          // (1)
  type U <: this.T     // (2)
}
class B extends A {    // (3)
  type T = A           // (4)
  type U = B           // (5)
}
```



- Idée : On doit pouvoir converger vers un même type S :

- L'exemple précédent est bien rejeté.

Transitivité par convergence

Pour formaliser la convergence,

Transitivité par convergence

Pour formaliser la convergence,

1. on "inline" la transitivité dans les règles de sous-typage :

$$(\textsc{S-Up})\ \frac{\begin{array}{c}\Gamma \vdash\ p : C \\ (\texttt{type}\ L <: T) \in C \\ \color{red}{\Gamma \vdash\ T[\texttt{this}\backslash p] <: S}\end{array}}{\Gamma \vdash\ p.L <: \color{red}{S}}$$

Transitivité par convergence

Pour formaliser la convergence,

1. on "inline" la transitivité dans les règles de sous-typage :

$$(\text{S-UP})\ \frac{\Gamma \vdash p : C \quad (\texttt{type}\ L <: T) \in C \quad \Gamma \vdash T[\texttt{this} \backslash p] <: S}{\Gamma \vdash p.L <: S}$$

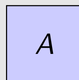
2. on restreint la règle de transitivité :

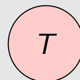
$$(\text{S-TRANS})\ \frac{\Gamma \vdash T <: S \quad \Gamma \vdash S <: U \quad S \prec T \quad S \prec U}{\Gamma \vdash T <: U}$$

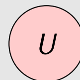
$S \prec T$: "il existe un chemin de T à S dans le graphe des symboles".

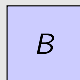
Cycles

```
class A {}
class B {}
class C {
  type U <: B       // (1)
  type T <: this.U  // (2)
  type U = this.T   // (3)
  type T = A        // (4)
}
```





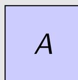


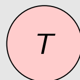


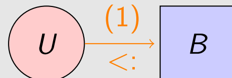
- Problème : programme bien formé, A sous-type de T et T sous-type de B (config. 2).
- Solution : interdire les cycles.

Cycles

```
class A {}
class B {}
class C {
  type U <: B       // (1)
  type T <: this.U // (2)
  type U = this.T   // (3)
  type T = A        // (4)
}
```



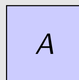


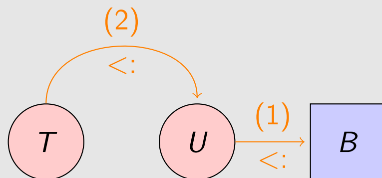


- Problème : programme bien formé, A sous-type de T et T sous-type de B (config. 2).
- Solution : interdire les cycles.

Cycles

```
class A {}
class B {}
class C {
  type U <: B        // (1)
  type T <: this.U // (2)
  type U = this.T  // (3)
  type T = A         // (4)
}
```

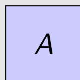


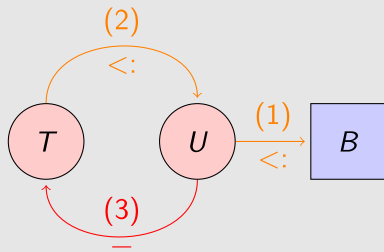


- Problème : programme bien formé, A sous-type de T et T sous-type de B (config. 2).
- Solution : interdire les cycles.

Cycles

```
class A {}
class B {}
class C {
  type U <: B       // (1)
  type T <: this.U // (2)
  type U = this.T  // (3)
  type T = A       // (4)
}
```

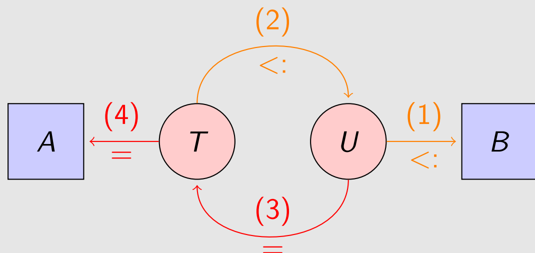




- Problème : programme bien formé, A sous-type de T et T sous-type de B (config. 2).
- Solution : interdire les cycles.

Cycles

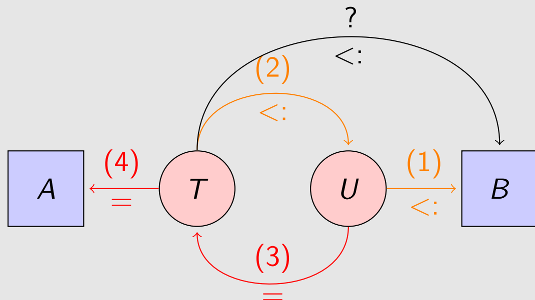
```
class A {}
class B {}
class C {
  type U <: B       // (1)
  type T <: this.U  // (2)
  type U = this.T   // (3)
  type T = A        // (4)
}
```



- Problème : programme bien formé, A sous-type de T et T sous-type de B (config. 2).
- Solution : interdire les cycles.

Cycles

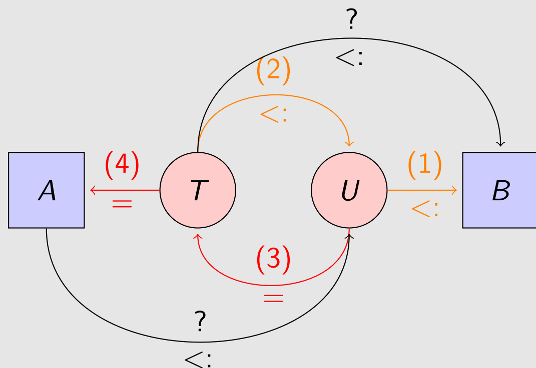
```
class A {}
class B {}
class C {
  type U <: B       // (1)
  type T <: this.U // (2)
  type U = this.T   // (3)
  type T = A        // (4)
}
```



- Problème : programme bien formé, A sous-type de T et T sous-type de B (config. 2).
- Solution : interdire les cycles.

Cycles

```
class A {}
class B {}
class C {
  type U <: B       // (1)
  type T <: this.U  // (2)
  type U = this.T   // (3)
  type T = A        // (4)
}
```



- Problème : programme bien formé, A sous-type de T et T sous-type de B (config. 2).
- Solution : interdire les cycles.

Solution simple pour empêcher les cycles

Exiger une relation $\prec$ sur les symboles de types et de classes qui soit :

1. bien-fondée (pas de suites infinies décroissantes).
2. compatible avec les définitions de types .
   Par ex. : `type T = this.U` implique $U \prec T$.
3. compatible avec les définitions de classes .
   Par ex. : `class B extends A` implique $A \prec B$.
4. telle qu'un symbole de classe et toujours plus petit qu'un symbole de type.

On peut faire des preuves par induction sur la relation $\prec$.

Rem : $\prec$ peut être inférée (pas d'annotations de la part de l'utilisateur), incrémentalement (compilation séparée).

Généralisation (temporaire) du sous-typage

- Dernier problème : On n'arrive **pas** à prouver la subject reduction par induction sur notre relation de typage.

$$\emptyset \vdash t : T \wedge t \longrightarrow t' \quad \Rightarrow \quad \emptyset \vdash t' : T$$

Généralisation (temporaire) du sous-typage

- Dernier problème : On n'arrive **pas** à prouver la subject reduction par induction sur notre relation de typage.

$$\emptyset \vdash t : T \wedge t \longrightarrow t' \quad \Rightarrow \quad \emptyset \vdash t' : T$$

- Solution : On généralise le typage en admettant la règle de transitivité sans contraintes.

$$\emptyset \vdash_{\mathsf{Gen}} t : T \wedge t \longrightarrow t' \quad \Rightarrow \quad \emptyset \vdash_{\mathsf{Gen}} t' : T$$

Généralisation (temporaire) du sous-typage

- Dernier problème : On n'arrive **pas** à prouver la subject reduction par induction sur notre relation de typage.

$$\emptyset \vdash t : T \wedge t \longrightarrow t' \qquad \Rightarrow \qquad \emptyset \vdash t' : T$$

- Solution : On généralise le typage en admettant la règle de transitivité sans contraintes.

$$\emptyset \vdash_{\mathsf{Gen}} t : T \wedge t \longrightarrow t' \qquad \Rightarrow \qquad \emptyset \vdash_{\mathsf{Gen}} t' : T$$

- Il reste à montrer que :

$$\emptyset \vdash t : T \qquad \Leftrightarrow \qquad \emptyset \vdash_{\mathsf{Gen}} t : T$$

Généralisation (temporaire) du sous-typage

- Dernier problème : On n'arrive **pas** à prouver la subject reduction par induction sur notre relation de typage.

$$\emptyset \vdash t : T \wedge t \rightarrow t' \quad \Rightarrow \quad \emptyset \vdash t' : T$$

- Solution : On généralise le typage en admettant la règle de transitivité sans contraintes.

$$\emptyset \vdash_{\text{Gen}} t : T \wedge t \rightarrow t' \quad \Rightarrow \quad \emptyset \vdash_{\text{Gen}} t' : T$$

- Il reste à montrer que :

$$\emptyset \vdash t : T \quad \Leftrightarrow \quad \emptyset \vdash_{\text{Gen}} t : T$$

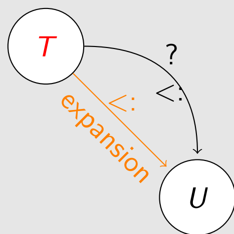
- C'est un corollaire direct du fait que la transitivité (sans contraintes) est admissible dans l'environnement vide.

$$\emptyset \vdash T <: S \wedge \emptyset \vdash S <: U \quad \Rightarrow \quad \emptyset \vdash T <: U$$

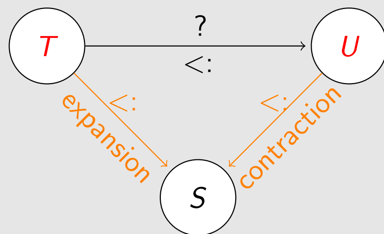
Implémentation des règles de sous-typage

- Question : Est-ce que la contrainte sur la transitivité réduit l'expressivité ?
- Réponse : En pratique, non ! Car un algorithme de typage utilise toujours la règle de transitivité restreinte.
- Transitivité générale : Juste pour faire passer la récurrence.

Recherche depuis `T` :



Test entre `T` et `U` :



Limitations

Par rapport à Scala, il manque à notre calcul :

- les préfixes plus complexes dans les types virtuels.

Listes d'entiers (ADT)

```
class ListModule {
  type T
  def head(xs: this.T): Int
  def tail(xs: this.T): this.T
  ...
}
```

```
class ListOperations {
  val lm: ListModule
  def sort(xs: this.lm.T): this.lm.T = ...
}
```

Limitations

Par rapport à Scala, il manque à notre calcul :

- les préfixes plus complexes dans les types virtuels.

Listes d'entiers (ADT)

```
class ListModule {
  type T
  def head(xs: this.T): Int
  def tail(xs: this.T): this.T
  ...
}
```

```
class ListOperations {
  val lm: ListModule
  def sort(xs: this.lm.T): this.lm.T = ...
}
```

- les raffinements dans les types classes : nécessaires pour encoder les types paramétriques (variants).

  `List[Int]` devient `List{type Elem = Int}`
  `List[+Int]` devient `List{type Elem <: Int}`

Limitations

Par rapport à Scala, il manque à notre calcul :

- les préfixes plus complexes dans les types virtuels.

Listes d'entiers (ADT)

```
class ListModule {
  type T
  def head(xs: this.T): Int
  def tail(xs: this.T): this.T
  ...
}
```

```
class ListOperations {
  val lm: ListModule
  def sort(xs: this.lm.T): this.lm.T = ...
}
```

- les raffinements dans les types classes : nécessaires pour encoder les types paramétriques (variants).

  `List[Int]` devient `List{type Elem = Int}`
  `List[+Int]` devient `List{type Elem <: Int}`

- les classes imbriquées.

Conclusion

- Les types virtuels sont au centre du système de typage de SCALA.
- La preuve d'un système avec types virtuels est loin d'être triviale, car il implique la combinaison de types dépendants des objets et de sous-typage.

Conclusion

- Les types virtuels sont au centre du système de typage de Scala.
- La preuve d'un système avec types virtuels est loin d'être triviale, car il implique la combinaison de types dépendants des objets et de sous-typage.
- On a donné une preuve de sûreté complètement formelle et commentée pour une extension de FJ avec types virtuels.
- Outils : restriction de la transitivité, relation bien fondée.

Conclusion

- Les types virtuels sont au centre du système de typage de SCALA.
- La preuve d'un système avec types virtuels est loin d'être triviale, car il implique la combinaison de types dépendants des objets et de sous-typage.
- On a donné une preuve de sûreté complètement formelle et commentée pour une extension de FJ avec types virtuels.
- Outils : restriction de la transitivité, relation bien fondée.
- Peut servir de base pour de futures extensions : généralisation des préfixes dans les types, types classes avec raffinements, classes imbriquées.
- Après la thèse : implémenter la preuve en COQ (fait), implémenter un vérificateur de types qui génère des preuves de typage.

SCALA est-il sûr ?

Pas totalement évident !

Contre-exemple dans scala-1.3.0.10 (Altherr/Cremet)

```
abstract class C { type T >: Int <: String }
val a: C = null
val x: a.T = 42
val y: String = x
```

- Symptôme : `y: String` et `y = x = 42`.
- Diagnostic : `C` n'est pas instanciable, pourtant `null` est une instance valide de `C`.
- Remède : interdire statiquement les bornes contradictoires.
- Question : y a-t-il d'autres contre-exemples ?

But et contributions

But de la thèse : valider le système de typage de SCALA de manière formelle.

But et contributions

But de la thèse : valider le système de typage de SCALA de manière formelle.

Contribution 1

La définition d'une sémantique formelle de SCALA, par traduction dans un calcul minimal basé sur les classes.

Motivation : La correction d'un système de typage ne peut être établie que par rapport à une description formelle de l'exécution d'un programme.

But et contributions

But de la thèse : valider le système de typage de Scala de manière formelle.

Contribution 1

La définition d'une sémantique formelle de Scala, par traduction dans un calcul minimal basé sur les classes.

Motivation : La correction d'un système de typage ne peut être établie que par rapport à une description formelle de l'exécution d'un programme.

Contribution 2

Une preuve formelle de correction des types virtuels.

Motivation : Les types virtuels sont le mécanisme de base pour l'abstraction de type en Scala.

Critères d'évaluation pour un système de typage

- Expressivité : le système de typage accepte des programmes non triviaux. *Informel*.
- Sûreté : les programmes acceptés s'exécutent correctement. *Démontrable formellement*.
- Décidabilité : l'ensemble des programmes acceptés est reconnaissable par un algorithme. *Formel*.

Implémentation : $\prec$ peut être...

1. inférée

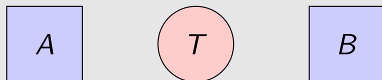
```
class Object {}

class A {
  type T                  ==> Object < T, A < T
         <: Object        ==> Object < T
  type U                  ==> Object < U, A < U
         <: Object        ==> Object < U
}
```

2. incrémentalement (compilation séparée)

```
class B                   ==> B < T, B < U
        extends A {       ==> A < B
  type T = this.U         ==> U < T
}
```

Bornes incompatibles

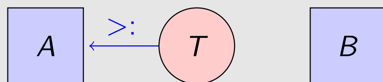
```
class A { }
class B { }
class C {
  type T >: A <: B
}
```



- Problème : S'il existe un objet v de type C, alors, avec nos règles, on peut déduire A sous-type de $v.T$ et $v.T$ sous-type de B.
- Heureusement : Il n'existe pas d'objet de type C vu que les bornes de T sont incompatibles.
- Plus de peur que de mal !

Bornes incompatibles

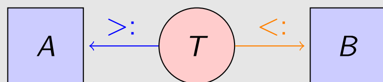
```
class A { }
class B { }
class C {
  type T >: A <: B
}
```



- Problème : S'il existe un objet v de type C, alors, avec nos règles, on peut déduire A sous-type de $v.T$ et $v.T$ sous-type de B.
- Heureusement : Il n'existe pas d'objet de type C vu que les bornes de T sont incompatibles.
- Plus de peur que de mal !

Bornes incompatibles

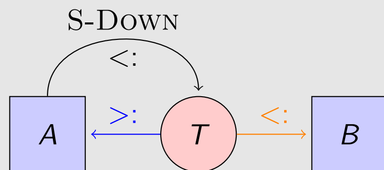
```
class A { }
class B { }
class C {
  type T >: A <: B
}
```



- Problème : S'il existe un objet v de type C, alors, avec nos règles, on peut déduire A sous-type de $v.T$ et $v.T$ sous-type de B.
- Heureusement : Il n'existe pas d'objet de type C vu que les bornes de T sont incompatibles.
- Plus de peur que de mal !

Bornes incompatibles

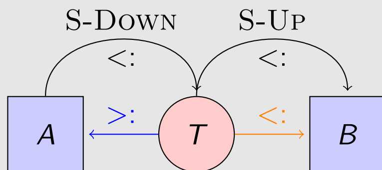
```
class A { }
class B { }
class C {
  type T >: A <: B
}
```



- Problème : S'il existe un objet v de type C, alors, avec nos règles, on peut déduire A sous-type de $v.T$ et $v.T$ sous-type de B.
- Heureusement : Il n'existe pas d'objet de type C vu que les bornes de T sont incompatibles.
- Plus de peur que de mal !

Bornes incompatibles

```
class A { }
class B { }
class C {
  type T >: A <: B
}
```



- Problème : S'il existe un objet v de type C, alors, avec nos règles, on peut déduire A sous-type de $v.T$ et $v.T$ sous-type de B.
- Heureusement : Il n'existe pas d'objet de type C vu que les bornes de T sont incompatibles.
- Plus de peur que de mal !