

# Vers la définition d'un préordre sur les programmes FeatherweightJava

Notes sur mon stage au LACL

Samy Avrillon

29 mars 2026

---

## Table des matières

# 1 Tentatives de définition d'un contexte sur Featherweight Java

## 1.1 Définitions et notations

On se base sur les notations issues de la définition du Featherweight Java [fjdef]. Donc  $e$  est la grammaire des expressions,  $L$  la grammaire des classes,  $\bar{x}$  dénote une suite potentiellement vide d'élément de type  $x$ ,  $<:$  désigne la relation de sous-typage,  $\longrightarrow$  désigne l'opérateur de réduction.

Un programme Featherweight Java est composé de deux parties : La première est une liste de définitions de classes, que le papier originel appelait *collection of class definitions*, mais que nous appellerons plutôt *Class Tables*. La seconde est une expression à évaluer (on peut la voir comme le *body* d'une fonction main d'un programme Java). Nous allons donc considérer qu'un programme FeatherweightJava  $P$  est un couple  $(\bar{L}, e)$ , et l'on notera  $CT(P) = \bar{L}$  et  $e_P = e$  lorsque nous manipulerons plusieurs programmes.

On note la grammaire à un trou  $h$  :

$$h := x \mid [.] \mid h.f \mid h.m(\bar{e}) \mid e.m(\bar{e}, h, \bar{e}) \mid \mathbf{new} \ C(\bar{e}, h, \bar{e}) \mid (C) \ h$$

On pourra souvent se ramener à la grammaire à plusieurs trous  $\tilde{h}$  :

$$\tilde{h} := x \mid [.] \mid \tilde{h}.f \mid \tilde{h}.m(\bar{\tilde{h}}) \mid \mathbf{new} \ C(\bar{\tilde{h}}) \mid (C) \ h \mid e$$

Puis on ajoute l'opération de *remplissage*  $h[g]$  pour  $h$  expression à trou et  $g$  expression classique, définie récursivement :

$$\begin{aligned} x[g] &= x \\ [.] [g] &= g \\ (h.f)[g] &= h[g].f \\ (h.m(\bar{e}))[g] &= h[g].m(\bar{e}) \\ (e.m(\bar{e}, h, \bar{e}))[g] &= e.m(\bar{e}, h[g], \bar{e}) \\ (\mathbf{new} \ C(\bar{e}, h, \bar{e}))[g] &= \mathbf{new} \ C(\bar{e}, h[g], \bar{e}) \\ ((C)h)[g] &= (C)h[g] \end{aligned}$$

## 1.2 Première définition naïve

La première définition, à l'instar de celle donnée pour le  $\pi$ -calcul pourrait ressembler à ça :

Un contexte est un élément de  $h$ .

Alors l'interprétation d'un programme dans un contexte est

$$C[P] = (CL(P), h[e])$$

Le problème, c'est que cette version des contextes est peu puissante. En effet, puisque nous avons juste des expressions, nous limitons à l'utilisation de la *class table* du programme. Et ça nous enlève tout une batterie de tests que l'on aurait pu autrement pû utiliser.

### 1.2.1 Problème du trou dégénéré

**Théorème 1** *Si la class table du programme testé permet de créer un trou dégénéré, c'est à dire un contexte à trou te que pour toute expression e, il existe une expression trouée h telle que  $\forall p \ h[p] \rightarrow^* e$ .*

*Alors  $P \leq Q \implies CT(P) \subseteq CT(Q)$*

Un exemple classique de contexte ayant un trou dégénéré est dans un programme dont la class table contient la classe Pair suivante :

```
class Pair extends Object {
  Object fst;
  Object snd;
  Pair(Object fst, Object snd) {
    super();
    this.fst = fst;
    this.snd = snd;
  }
}
```

Alors si on considère le contexte suivant :

$$C = \text{new Pair}(e, [.]).\text{fst}$$

On aura pour toute expression e :

$$C[p] = \text{new Pair}(e, p).\text{fst} \rightarrow^* e$$

Le théorème se prouve en construisant des expressions qui dépendent explicitement de la structure des classes. Par exemple, `new A()` ne se résout en une valeur seulement si la *class table* contient une classe A sans attributs. On peut aussi par exemple imposer le type des attributs de C ainsi que l'existence d'une méthode C.m prenant un argument de type A avec l'expression suivante :

$$\text{new C}(\text{new A}(), \text{new B}()).\text{m}(\text{new A}())$$

**C'est un problème** par ce que deux programmes ne peuvent pas utiliser des classes internes différentes. Par exemple, un programme de tri d'une liste qui utiliserait une classe Map et un autre qui utiliserait Tree ne pourraient être comparés, quand bien même leurs fonctionnements pourraient paraître similaire à nos yeux. On a le même problème sur le nom des classes « internes ».

**Ce n'est pas un problème** car sans notion d'access control, un utilisateur peut utiliser toutes les classes internes. Donc la structure interne du programme a de l'importance sur ce qu'on peut modifier du programme.

**Une solution** serait de redéfinir le préordre pour ne considérer que les classes sur les parties des class tables qui sont en commun. Par contre, on ne résout pas le problème de la dépendance au nom, si deux programmes définissent une même classe MaClasse, doit-on imposer pour les programmes soient équivalents que les deux « implémentations » le soient ?

**Une autre solution** pourrait être de comparer uniquement avec des contextes qui « utiliseraient » (notion à définir) une partie spécifique de la Class Table qui serait alors comparée. On retrouve l'idée d'interface présente dans le Java complet (ou d'autres versions de FJ [liquori\_fjInterfaces]).

**Une dernière solution** serait d'ailleurs d'ajouter les access control au featherweight java, c'est à dire de pouvoir noter les classes comme public, private, package-protected ou protected, en restreignant directement le typage des sexpressions.

### 1.3 Seconde définition naïve

Un contexte est une class table  $\bar{L}$  et un contexte à trou  $h$ .

Alors  $C[P] = (CT(P) + \bar{L}', h[\bar{L}'/\bar{L}][e_P])$  où  $\bar{L}'$  est une *class table*  $\alpha$ -convertible en  $\bar{L}$  tel qu'aucun nom de classe ne coïncide avec ceux de  $CT(P)$ .

On définit un  $\alpha$ -renommage de classe dans une class table comme l'opération de changer un nom de classe en un autre nom, en changeant toutes les occurrences de ce nom (**new**  $C(\dots)$ ,  $(C)e$ , **extends**, définition d'attributs, de paramètres de méthodes et de type de retour de méthode), en s'assurant que le nom n'était pas utilisé.

Deux class tables sont alors  $\alpha$ -convertibles si on peut passer de l'une à l'autre avec un nombre fini d' $\alpha$ -renommages, à l'instar de ce qui est fait dans le  $\pi$ -calcul [picalcul\_sangiorgi].

On a alors forcément un trou dégénéré (parce que on peut ajouter la classe Paire nous-même)

### 1.4 Troisième définition naïve

Un contexte est une class table  $\bar{L}$  et une expression  $e_C$ .

Alors  $C[P] = (CT(P) + \bar{L}' + Main'', e_C[CT'/CT][Main''/Main])$  où  $\bar{L}'$  est obtenu de la même manière que précédemment et  $Main$  est la class table suivante :

```
class Main extends Object {
  Main() {
    super();
  }
  Object main(Object  $\bar{x}$ ) {
    return e;
  }
}
```

où  $\bar{x}$  est l'ensemble des variable libres de  $e_P$ .

On sent que l'on veut tester l'utilisation d'une méthode plutôt que d'une expression, ce qui est en adéquation avec le paradigme orienté objet, mais ces contextes restent toujours aussi faillibles.

On peut forcer un peu plus l'utilisation du main, en construisant le contexte dans sa version 3.1 :  $C = \bar{L}$  et  $\bar{e}_C$  interprété en

$$C[P] = (CT(P) + \bar{L}' + Main'', \mathbf{new} Main().main(\overline{e_C[CT'/CT][Main''/Main]}))$$

### 1.5 Quatrième définition naïve

La première méthode naïve est utile pour faire des équivalences entre des programmes de même class table. Mais on pourrait faire mieux.

On définit déjà un préordre sur les class tables :

$$CTA \prec CTB \iff \forall e \forall v (CTA, e) \Downarrow v \implies (CTB, e) \Downarrow v$$

On peut ainsi utiliser la méthode du contexte pour créer le préordre suivant sur les programmes :

$$P \prec Q \iff \left| \begin{array}{l} CT(P) \prec CT(Q) \\ \forall h (CT(P), h[e_P]) \Downarrow v \implies (CT(Q), h[e_Q]) \Downarrow v \end{array} \right.$$

On peut aussi encore augmenter le contexte en le définissant avec sa propre class table :

Un contexte  $C$  est alors une  $CT \bar{L}$  et une expression trouée  $h$ .

L'interprétation est alors  $C[P] = (CT(C), h[e_P])$  si  $CT(P) \prec CT(C)$ . On dit alors que  $C$  convient à  $P$ , noté  $C \triangleleft P$ .

On a alors un nouveau pré-ordre sur les programmes :

$$P \prec Q \iff \forall C (C \triangleleft P \text{ et } C \triangleleft Q) \implies (\forall v C[P] \Downarrow v \implies C[Q] \Downarrow v)$$

**Gros problème cependant** si les deux programmes comparées n'ont aucun contexte qui leur convienne à tous les deux. On est alors bloqués dans une situation d'universalité du vide. Par exemple les deux class table suivantes, notées  $CT_A$  et  $CT_B$  :

<pre> class A extends Object {   A() {     super();   } } class B extends Object {   B() {     super();   } } class Get extends Object {   A a;   B b;   Get(A a, B b) {     super();     this.a = a;     this.b = b;   }   Object get(){     return this.a;   } } </pre>	<pre> class A extends Object {   A() {     super();   } } class B extends Object {   B() {     super();   } } class Get extends Object {   A a;   B b;   Get(A a, B b) {     super();     this.a = a;     this.b = b;   }   Object get(){     return this.b;   } } </pre>
---	---

Alors en notant  $e_A = (A)(\text{new Get}(\text{new A}(), \text{new B}()).get())$  et  $e_B = (B)(\text{new Get}(\text{new A}(), \text{new B}()).get())$  on a

$$(CL_A, e_A) \Downarrow \text{new A}(); (CL_A, e_B) \nDownarrow; (CL_B, e_A) \nDownarrow; (CL_B, e_B) \Downarrow \text{new B}()$$

Donc aucun contexte ne peut satisfaire les deux class tables. Ce qui est problématique !

On peut déjà chercher à simplifier le programme, afin d'enlever les opérations inutiles quand elles ne sont pas au cœur des opérations. Par exemple, dans le cas d'un programme totalement légitime, l'ajout de ces deux classes annulerai tout le travail, même si elles ne sont pas utilisées.

On définit qu'un programme  $P'$  est plus simple que  $P$ , noté  $P' < P$  par trois axiomes :

1.  $CT(P') < CT(P)$
2.  $e_{P'} = e_P$
3.  $\forall v P \Downarrow v \implies P' \Downarrow v$

## 2 Définir l'équivalence sur les expressions

Nous avons déjà défini le préordre suivant sur les Class Tables :

$$CTA < CTB \iff \forall e \forall v (CTA, e) \Downarrow v \implies (CTB, e) \Downarrow v$$

Mais cette définition pose deux problèmes :

- On n'a toujours pas de préordre sur les programmes entiers, ni sur les expressions.
- Cette définition sémantique est difficile à vérifier, il nous faudrait une caractérisation de cette équivalence plus simple à vérifier en pratique.

### 2.1 Une équivalence naïve

Si l'on souhaitait comparer deux expressions de but en blanc, dans une class table donnée.

La version de l'expression trouée est superflue, en effet, on a pour toute class table  $CT$  et toute paire d'expressions  $e$  et  $e'$  :

$$(\forall h (CT, h[e]) \Downarrow v \implies (CT, h[e']) \Downarrow v) \quad \text{ssi} \quad ((CT, e) \Downarrow v \implies (CT, e') \Downarrow v)$$

Le sens direct se fait en particulierisant à  $h = []$  et le sens indirect se fait en appliquant les règles de réduction de congruence qui donnent

$$\forall (e, e') \forall h \quad e \rightarrow e' \implies h[e] \rightarrow h[e']$$

On peut alors essayer de dire que deux expressions ont le même fonctionnement sous toute class table, c'est à dire vérifier que  $\forall CT ((CT, e) \Downarrow v \implies (CT, e') \Downarrow v)$ .

On va essayer de montrer que ce n'est pas vrai, en prouvant la chose suivante pour toute paire de formules  $e$  et  $e'$  qui soient cohérentes (c'est à dire  $\exists CT \exists v (e, CT) \Downarrow v$ , ce qui permet de s'assurer que l'on essaie pas de démentir Faux  $\implies$  Faux, par exemple si  $e$  vaut **new**  $A().field$ ) :

$$\exists CT \quad (e, CT) \Downarrow v \wedge (e', CT) \not\Downarrow v$$

Hélas, ce résultat est faux, comme en témoignant les deux expressions suivantes :

**new**  $Pair(\mathbf{new} A(), \mathbf{new} B()).fst$

et

**new**  $Pair(\mathbf{new} Pair(\mathbf{new} A(), \mathbf{new} B()).fst, \mathbf{new} B()).fst$

Nous n'avons alors que trois types de classes tables : Celles qui ne compilent pas (Alors  $\perp \implies \perp$ ), Celles où l'attribut *fst* est donné en premier dans le constructeur de *Pair* (Alors  $e_1 \Downarrow \mathbf{new} A()$  et  $e_1 \Downarrow \mathbf{new} A()$ ) et celles où l'attribut *fst* est donné en second dans le constructeur de *Pair* (Alors  $e_1 \Downarrow \mathbf{new} B()$  et  $e_2 \Downarrow \mathbf{new} B()$ ).

Pire encore, on a que pour toutes formules  $e$  et  $e'$ ,  $\phi(e)$  et  $\phi(e')$  sont équivalents en ce sens avec  $\phi$  défini comme

$$\phi(e) = \mathbf{new} \text{ Pair}(\mathbf{new} \text{ Pair}(\mathbf{new} \text{ A}(), e).fst, \mathbf{new} \text{ B}()).fst$$

Cela est destructeur puisque l'on obtient des équivalences entre des expressions qui sont absurdes, par exemple, si  $e = \mathbf{new} \text{ T}(\mathbf{new} \text{ T}())$  qui est une valeur, mais qu'on ne peut mettre dans aucune methode, parce que la class table ne pourrait être cohérent.

Nous allons donc définir la *consistance* d'une liste d'expression FJ  $\bar{e}$  :

$$e_1, e_2, \dots, e_k \text{ sont consistants} \quad \text{ssi} \quad \exists CT \forall i \in \llbracket 1, k \rrbracket \exists T \emptyset \vdash_{CT} e : T$$

## 2.2 Preuve de l'inefficacité sur un ensemble d'expressions plus restreintes

Nous pouvons tout de même essayer de prouver le résultat pour des expressions plus restreintes. Par exemple, des expressions qui ne sont composées que de **new** et d'appels à méthode (on peut imaginer remplacer les field access par des getters).

Je vais maintenant expliquer comment obtenir une Class table qui prouve le théorème.

Nous allons créer une class table telle que la méthode  $m$  de la classe  $C$  aie pour type de retour  $\Phi(m, C)$  qui soit un nom de classe non utilisé. Par exemple, on choisit un mot qui n'est préfixe d'aucun nom de classe utilisé (pour l'exemple  $F$ ), on choisit aussi un délimiteur une chaîne de caractères inutilisées dans les noms de méthode (pour l'exemple  $@$ ), et on note alors le type de retour de la méthode `method` dans la classe `Classe` par `Fmethod@Classe`. on peut alors déterminer à partir du nom de la classe d'un objet la méthode et la classe associée.

Ensuite, pour chaque occurrence d'un appel à méthode  $e.m(\bar{x})$  on détermine d'abord la classe de  $e$ . C'est assez simple récursivement, puisque si  $e$  est de la forme **new**  $C(\bar{x})$ , alors  $e$  est de classe  $C$ , et si  $e$  est de la forme  $e'.m'(\bar{x})$ , alors on détermine récursivement la classe  $C'$  de  $e'$ , puis on annonce que  $e$  a le type  $\phi(m', C')$ , soit dans notre exemple  $Fm'@C'$ .

Une fois que l'on a déterminé la classe de  $e$ , on peut ajouter à  $C$  une méthode  $m$  définie ainsi :

```
Fm@C m(Object x1,...,Object xk){
  return new Fm@C(this,x1,...,xk);
}
```

Si la classe  $C$  n'existe pas et que c'est une classe de la forme  $Fm@C$ , alors on peut la créer ainsi.

Classe	Méthode	Types paramètres	→	Type but
A	.quoi	$\emptyset$	→	Fquoi@A
Fquoi@A	.qui	{Fquel@B}	→	Fqui@Fquoi@A
B	.quel	{Fquoi@B}	→	Fquel@B
B	.quoi	{Fquoi@A}	→	Fquoi@B
A	.quoi	$\emptyset$	→	Fquoi@A

FIGURE 1 – Structures de classes obtenues par l'exemple

```

class Fm@C extends Object {
    C obj;
    Object x1;
    :
    Object xk;
    Fm@C(C obj, Object x1, ..., Object xk){
        super();
        this.obj=obj;
        this.x1=x1;
        :
        this.xk=xk;
    }
}

```

Appliquons maintenant à l'expression  $e$  suivante :

`new A().quoi().qui(new B().quel(new B().quoi(new A().quoi())))`

Nous obtenons les structures de classes ?? (dans l'ordre de lecture des applications de méthodes).

On remarque que la première et dernière ligne sont égales. C'est normal, c'est parce que l'on appelle deux fois la même fonctions (On a une contrainte structurelle qui nous impose que ce soit la même fonction). Et puisque le programme testé est consistant, elle est appelée avec le même nombre d'arguments (possiblement des types différents, mais nous ne regardons pas le type et castons tout en Object).

On obtient alors la class table ?? et ??

Alors notre exemple se résout en l'objet suivant : (les **new** ont été enlevés pour plus de lisibilité)

$e \rightarrow^* \text{Fqui@Fquoi@A}(\text{Fquoi@A}(A), \text{Fquel@B}(B, \text{Fquoi@B}(B, \text{Fquoi@A}(A))))$

On remarque que l'on peut reconstruire la formule à partir de la valeur finale. Il suffit de remplacer **new** Fmethod@Class( $e_0, e_1, \dots, e_k$ ) par  $e_0.\text{method}(e_1, \dots, e_k)$  puis d'appliquer récursivement sur les  $e_i$ .

Mais alors, étant donné deux expressions  $e$  et  $e'$  consistantes entre elles telles que  $\forall C (CT, e) \Downarrow v \wedge (CT, e') \Downarrow v$ . Alors, en particulier pour la class table que nous venons de créer, on a que les objets finaux sont égaux. Or, il est possible d'obtenir la formule qui a initié la valeur, donc les deux expressions qui n'utilisent pas les classes rajoutées, c'est à dire  $e$  et  $e'$  sont nécessairement égales.

Nous pouvons aussi tester avec la restriction symétrique : Comment déterminer si deux programmes sont équivalents lorsqu'ils ne sont composés uniquement de **new** et de *field access*?



```

class A extends Object {
    A() {
        super();
    }
    Fquoi@A quoi(){
        return new Fquoi@A(this);
    }
}

```

```

class B extends Object {
    B() {
        super();
    }
    Fquel@B quel(Object x){
        return new Fquel@B(this,x);
    }
    Fquoi@B quoi(Object x){
        return new Fquoi@B(this,x);
    }
}

```

FIGURE 2 – Classe A et B donnés pour l'expression exemple

Dans cet autre cas simple, on n'utiliserai aucune méthode des éventuelles Class Tables, donc deux class tables se différenciant uniquement sur leurs méthodes produiront le même résultat. La même chose pour les noms de class inutilisés, ils sont .... inutilisés et donc ne changent pas le résultat si ils apparaissent ou disparaissent.

On se retrouve avec un nombre restreint de classes, il suffit de prendre la class table dans laquelle la formule est typable, de supprimer les classes inutilisées et les méthodes, puis de permuter l'ordre des attributs pour chaque classe. Il y a un nombre fini de combinaisons (encore moins si on ne considère pas les attributs non considérés), que l'on peut tester facilement (voir le contre-exemple du début, prouvé avec cette méthode).

Mais quid des cas un peu plus intermédiaires ?

### 3 Des tests plus ciblés

Une première idée afin de faire des tests plus « ciblés » est de définir arbitrairement un jeu de méthodes, classes, fields, sur les quelles les programmes seront comparés.

#### 3.1 Définition des structures

Nous allons tout d'abord définir une grammaire des *Test interfaces* ou « interfaces de tests » à la ???. Un exemple de tests est donné à la ?? où l'on définit des classes Int, Frac, Number et RichInt tel qu'on puisse appeler certaines méthodes sur elles. On impose aussi de pouvoir construire les object Frac avec deux paramètres, et on impose aussi un attribut nommé value dans la classes RichInt.

```

class Fquoi@A extends Object {
  A obj;
  Fquoi@A(A obj) {
    super();
    this.obj = obj;
  }
  Fqui@Fquoi@A qui(Object x){
    return new Fqui@Fquoi@A(this,x);
  }
}
class Fqui@Fquoi@A extends Object {
  Fquoi@A obj;
  Object x;
  Fqui@Fquoi@A(Fquoi@A obj, Object x) {
    super();
    this.obj = obj;
    this.x = x;
  }
}
class Fquel@B extends Object {
  B obj;
  Object x;
  Fquel@B(B obj, Object x) {
    super();
    this.obj = obj;
    this.x = x;
  }
}
class Fquoi@B extends Object {
  B obj;
  Object x;
  Fquoi@B(B obj, Object x) {
    super();
    this.obj = obj;
    this.x = x;
  }
}
}

```

FIGURE 3 – Classes artificielles pour l'expression exemple

TI := C : C m(barC)	TSG-METH
C {barC barf}	TSG-ATTR
C (barC (?   f))	TSG-CSTR
C :> C	TSG-CAST

FIGURE 4 – Grammaire des test sets

```

Int {}
Int : Int suivant(Int)
Int : Int add(Int,Int)

Frac(Int numerateur, Int denominateur)
Frac : Frac inverted()
Frac : Int floor()

Number {}
Frac <: Number
Int <: Number

RichInt {Int value}
RichInt : Int getInt()
RichInt <: Int

```

FIGURE 5 – Exemple de set de tests

Afin de valider un tests set, il faut simplement que :

- Chaque nom de classe est défini au plus une fois par une règle de type ?? et ??.
- Chaque nom de methode est défini au plus une fois par nom de classe par une règle de type ??.
- Les noms de classes utilisés sont définis.

On pourrait se questionner sur la structure donnée aux règles ?? et ?? . La différence fondamentale entre les deux est que la première n'autorise pas à appeler **new** C(...) alors que la seconde l'autorise. La première autorise seulement l'accès aux champs de la classe. On pourrait alors critiquer que cette façon de définir les règles ne permet pas d'autoriser la création d'un objet avec **new** et d'autoriser certains *field access* sans en imposer l'ordre. Pourtant, autoriser ces deux constructions impose que deux *class tables* qui seraient équivalentes auraient le champ à la même position. Par exemple, on peut construire une expression simple **new** C(**new** P1(), **new** P2(), **new** P3(), ...).field où P1, P2, P3 sont des classes créées pour l'occasion, qui ont des paramètres cohérents, etc... Alors cette expression se réduira en la même valeur dans deux *class table* si et seulement si l'attribut field est assigné au même indice dans le constructeur de C dans les deux class tables.

La troisième règle permet d'éviter des définitions « implicites » de classes, en suivant la convention suivie par la première définition du Featherweight Java [fjdef]. On peut sinon ajouter la règle A{} pour chaque nom de classe utilisé sans être défini.

Nous allons aussi rajouter trois règles, qui ne sont pas strictement nécessaires, mais qui permettent de restreindre les tests sets aux tests sets « intéressants », en enlevant certaines contraintes qui feraient apparaître des méthodes ou objets inutilisables.

- La relation obtenue par cloture réflexive et transitive des règles de type ?? est un bon ordre (en ajoutant la classe virtuelle Object)
- Dans les occurrences des règles ?? et ??, la relation « est le type d'un attribut » induit un bon ordre.

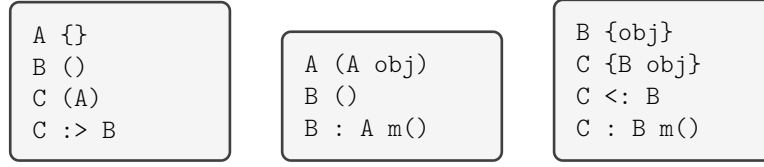


FIGURE 6 – Exemples de *test sets* implémentés par aucune *class table* sans cast

— Pour chaque règle ??  $C :> B$ , alors, la propriété adéquate est vérifiée, suivant les règles qui

ont défini C et B selon le tableau suivant :	$\frac{C\{\bar{F} \ \bar{f}\}}{C(\bar{F} \ ?f)}$	$B\{\bar{G} \ \bar{g}\}$	$f_i = g_j \implies F_i = G_j$	$B(\bar{G} \ ?g)$	$\frac{\exists ?f' \ \exists ?g'}{\{ \bar{F} \ (\bar{?f} \oplus \ ?f') \} \subset \{ \bar{G} \ (\bar{?g} \oplus \ ?g') \}}$

La première règle semble cohérente, on peut imaginer simplement deux règles  $A <: B$  et  $B <: A$  qui n'auraient pas de sens ensemble.

La troisième règle force les contraintes sur les types et noms des attributs à être cohérentes.

On aimerait dire qu'il existe toujours au moins une *class table* qui implémente une *test interface* qui respecte toutes ces règles, mais hélas, ce n'est pas vrai, si l'on considère de manière naïve la relation « est le type d'un attribut » induit par la règle 2, voir troisième exemple de la ??

Cependant, on peut toujours (si la première et la troisième règle sont vérifiées) créer une *class table* bidon qui ne compile jamais. Pour cela, il suffit d'implémenter naïvement chaque class, sauf que nous remplaçons chaque occurrence de `extends Object` par `extends SObject` (où `SObject` est un nom de classe inutilisé). On crée la classe `SObject` avec un constructeur ne prenant pas de paramètre. Alors, on peut implémenter n'importe quelle méthode du type  $\bar{M} \rightarrow T$  par l'expression  $(T) \text{ new } SObject()$ . L'expression est bien typable en  $T$  puisque `SObject` est une superclasse de tous les types (sauf `Object`, auquel cas renvoyer simplement `new SObject()`). Mais bien sûr, ces méthodes ne pourront jamais renvoyer une valeur car le cast sera toujours impossible.

Maintenant que nous avons notre grammaire, on peut définir plusieurs choses avec elles. Déjà, on va faire une relation  $CT \triangleright TI$  pour dire que la *class table*  $CT$  implémente la *test interface*  $TI$ , présentée à la ??. L'implémentation est la plus tolérante possible quant au typage notamment des méthodes, car on ne veut pas comparer les *class tables* sur leur typage, mais bien sur leur fonctionnement, le typage reste un garde-fou qui nous empêche de tester tout et n'importe quoi et d'obtenir une relation inexploitable. Typiquement, les deux *class tables* présentées ?? implémenteront toutes deux la *test interface* jointe, et seront donc équivalentes.

Nous allons donc maintenant essayer de typer une expression et une *class table* sous une *test interface*, dire que une expression  $e$  avec sa *class table* associée  $CT$  est « valide » dans un *test interface*  $TI$  associé, que l'on note  $TI, CT \vdash e : T$  (pour bien différencier de la règle de typage de Featherweight Java, notée  $\vdash$ ).

La présence d'une *class table* associée à l'expression est nécessaire car il existe deux *class tables* implémentant la même *test interfaces* qui ne sont pas différenciables par des expressions seules, mais qui le sont lorsque l'on ajoute un *class table* (voir exemple ??).

Les règles de typage sont obtenues en modifiant quelque peu les règles de typage de Featherweight java. La première étape est de définir les deux applications *mtyp* à partir des règles de type 1 et des déclarations de méthodes, et *fields* qui s'obtient à partir des règles deux et trois et des champs présents dans la *class table*. On rajoute enfin une application `construct(C)` qui est défini avec deux règles, l'une à partir de la règle trois et l'autre à partir des constructeurs réels (voir ??).

Nous pouvons alors définir la relation de typage d'une expression sous une *class table* et une

$$\begin{array}{l}
\frac{\text{mtype}_{CT}(\mathbf{m}, C) = \bar{E} \rightarrow D}{CT \triangleright [C : D \text{ m}(\bar{E})]} \quad (1) \\
\\
\frac{\bar{E} \bar{f} \subset \text{fields}_{CT}(C)}{CT \triangleright [C \text{ } \{\bar{E} \bar{f}\}]} \quad (2) \\
\\
\frac{\exists \bar{f}' \bar{f}\bar{0} = \bar{f} + \bar{f}' \wedge \bar{E} \bar{f}\bar{0} = \text{fields}_{CT}(C)}{CT \triangleright [C \text{ } (\bar{E} \bar{f})]} \quad (3) \\
\\
\frac{C <:_{CT} D}{CT \triangleright [C <: D]} \quad (4)
\end{array}$$

FIGURE 7 – Grammaire des test sets

```

A {}
B {}
Static ()
Static : A get()

```

```

class A extends Object {
  A() {
    super();
  }
}
class B extends A {
  B() {
    super();
  }
}
class Static extends Object {
  Static() {
    super();
  }
  A get(){
    return new B();;
  }
}

```

```

class A extends Object {
  A() {
    super();
  }
}
class B extends A {
  B() {
    super();
  }
}
class Static extends Object {
  Static() {
    super();
  }
  B get(){
    return new B();;
  }
}

```

FIGURE 8 – Exemple de classes justifiant la tolérance dans la définition de  $\triangleright$

```
B ()  
Static ()  
Static : B get(B)
```

```
class B extends Object {  
    B() {  
        super();  
    }  
} class Static extends Object {  
    Static() {  
        super();  
    }  
    B get(B in){  
        return in;  
    }  
}
```

```
class B extends Object {  
    B() {  
        super();  
    }  
} class Static extends Object {  
    Static() {  
        super();  
    }  
    B get(B in){  
        return new B();  
    }  
}
```

```
class A extends B {  
    A() {  
        super();  
    }  
} new State().get(new A())
```

FIGURE 9 – Exemple montrant l'utilité de la *class table* de test

$$\frac{\mathcal{C}(\overline{D} \ \overline{f})\{\dots\} \in CT}{\text{construct}(C) = \overline{D}} \quad (5)$$

$$\frac{\mathcal{C}(\overline{D} \ \overline{?f}) \in TI}{\text{construct}(C) = \overline{D}} \quad (6)$$

FIGURE 10 – Application  $\text{construct}(C)$

*test interface* avec des règles d'inférence présentées ?? . On ajoute aussi toutes les règles de typage d'expression présentées dans la définition du FJ Java [fjdef], en enlevant la règle T-SCAST qui n'ajoute pas grand chose ...

**Le théorème de validité du typage** que nous allons essayer de prouver s'énonce ainsi : Soit une *test interface*  $TI$ , une *class table*  $CT$  et un test  $(CT_t, e)$  vérifiant

- $CT \triangleright TI$
- $CT_t$  OK IN  $TI$
- $TS, CT_t \Vdash e : D$

Alors il existe  $C$  tel que  $CT \oplus CT_t \vdash e : C$  et  $C <: D$ .

Cette dernière condition est nécessaire à cause de la tolérance de l'opérateur  $\triangleright$ , comme est démontré par l'exemple ??.

## 4 Équivalence de méthodes

**Définition 1** Deux méthodes de même nom  $\text{Class.method}$  dans deux *class tables* sont en pré-ordre lorsque pour toute valeur  $V$ , valeurs  $\overline{v}$  et valeurs  $\overline{p}$  :

$$(CT_1, e) \Downarrow V \implies (CT_2, e) \Downarrow V$$

avec  $e = \mathbf{new} \ \text{Class}(p_1, p_2, \dots, p_n).method(v_1, v_2, \dots, v_m)$

On note alors  $CT_1 \prec[\text{Class.method}] CT_2$

Alors, nous souhaiterions prouver le théorème suivant :

**Théorème 2** Soient deux *class tables*  $CT_1$  et  $CT_2$  et une *test interface*  $TI$ ,

$$CT_1 \prec_{TS} CT_2 \iff \forall [\text{Class} : D \ \text{meth}(\overline{E})] \in TI \quad CT_1 \prec[\text{Class.meth}] CT_2$$

### 4.0 Ajout de null à Featherweight Java

#### 4.1 Lemme des arguments finis

Pour un ensemble  $\overline{CT}$ , on note  $\bigwedge \overline{CT}$  l'ensemble des valeurs qui sont valides dans chaque *class table* de  $\overline{CT}$ .

$\frac{x \in \Gamma}{TI, CT, \Gamma \Vdash x : \Gamma(x)}$	TI-VAR
$\frac{TI, CT, \Gamma \Vdash e : C \quad C <: D \quad [D(\dots, E \ f, \dots)] \in TI}{TI, CT, \Gamma \Vdash e.f : E}$	TI-FIELD
$\frac{TI, CT, \Gamma \Vdash e : C \quad TI, CT, \Gamma \Vdash \overline{e'} : \overline{E} \quad C <: D \quad \overline{E} <: \overline{G} \quad [D : R \ m(\overline{G})] \in TI}{TI, CT, \Gamma \Vdash e.m(\overline{e'}) : R}$	TI-INVK
$\frac{TI, CT, \Gamma \Vdash \overline{e} : \overline{E} \quad \overline{E} <: \overline{G} \quad [D(\overline{G})] \in TI}{TI, CT, \Gamma \Vdash \mathbf{new} \ D(\overline{e}) : D}$	TI-NEW
$\frac{TI, CT, \Gamma \Vdash e : C \quad C <:_{TI} D}{TI, CT, \Gamma \Vdash (D)e : D}$	TI-DCAST
$\frac{TI, CT, \Gamma \Vdash e : C \quad D <:_{TI} C \quad C \neq D}{TI, CT, \Gamma \Vdash (D)e : D}$	TI-UCAST

FIGURE 11 – Typage des expressions avec un test set et une class table

Soient  $CT_i$  des class tables.

Soient  $\overline{d}$  des expressions typables dans chaque  $CT_i$  telles que pour toute sous-expression  $\varepsilon$  de  $d_i$ ,  $\forall i, j \ (CT_i, \varepsilon) \Downarrow v \iff (CT_j, \varepsilon) \Downarrow v$

Alors il existe une famille d'ensembles de variables  $\overline{V}$  indexée par les  $\overline{d}$ , dont les variables sont dans  $\bigwedge CT_i$  telle que

$$[\overline{d}/\overline{x}] e \rightarrow^* V \iff \forall v_i \in V_i \quad [\overline{v}/\overline{x}] e \rightarrow^* V$$

$\overline{V}$  dépend donc de  $\bigwedge CT_i$  et de  $d$ .

#### 4.1.1 Preuve de la confluence de la réduction en Featherweight Java

Nous allons dans cette section essayer de démontrer que la relation  $\rightarrow$  de réduction de Featherweight Java est *confluente*, c'est à dire que

$$\forall e, e_1, e_2 \quad e_1 \leftarrow e \rightarrow^* e_2 \implies (\exists e' \quad e_1 \rightarrow^* e' \leftarrow e_2)$$

La relation dénote ici la réduction sous une class table quelconque fixée.

On peut se contenter de montrer la semi-confluence, c'est à dire

$$\forall e, e_1, e_2 \quad e_1 \leftarrow e \rightarrow^* e_2 \implies (\exists e' \quad e_1 \rightarrow^* e' \leftarrow e_2)$$

Si  $e \rightarrow e_1$ , c'est qu'il y a un nœud qui s'est fait réduire de la forme  $e.f$ ,  $e.meth(\dots)$  ou  $(C) \ e$  où  $e$  est de la forme  $\mathbf{new} \ D(\dots)$ . On va procéder à un *marquage* de ce nœud afin de pouvoir l'observer dans  $e_2$ . Le marquage perdurera le long de la réduction  $e \rightarrow^* e_2$ . Notez que



```

E {}
D ()
D : E m()

```

```

class E extends Object {
    E() {
        super();
    }
}
class F extends E {
    F() {
        super();
    }
}
class F extends Object {
    F() {
        super();
    }
    F m(){
        return new F();
    }
}

```

$e = \text{new } D().m()$

$TI \Vdash e : E \quad CT \vdash e : F \quad CT \triangleright TS$

FIGURE 12 – Exemple de la nécessité de la seconde clause du théorème

le marquage se démultiplie si un argument il est dans un argument d'une méthode qui se fait déplier et qu'il est utilisé plusieurs fois, et qu'il est détruit si la réduction s'effectue sur ce nœud.

Ensuite, pour chaque expression du chemin  $e \rightarrow^* e_2$ , on remplace chaque nœud marqué par sa version réduite par application de la même règle que pour  $e \rightarrow e_1$ . On obtient un chemin  $e_1 \rightarrow^* e'$  car une réduction ne modifie que le nœud qu'elle réduit plus éventuellement un nœud **new** associé, or, une expression sous la forme **new**  $C(\dots)$  ne pourra être réduite que en une expression de la même forme.

Enfin, nous avons  $e_2 \rightarrow e'$  car il suffit de réduire séquentiellement chaque nœud marqué dans  $e_2$  par la même opération que pour  $e \rightarrow e_1$ , et nous obtenons par définition  $e'$ .

#### 4.1.2 Obtention des ensembles de valeurs $V$

Pour créer l'ensemble de valeurs, nous allons nous restreindre à une seule *class table* (la relation de réduction est donc simplement notée  $\rightarrow$ ) et une seule expression  $e \in \bar{d}$ .

Nous allons déjà définir une application appelée *varnull* qui transforme une expression en valeur en transformant les nœuds selon le tableau suivant :

Remplacer	Par
$e.f$	$\text{null}(D)$ si $(D, f) \in \text{fields}(C)$ et $\vdash e : C$
$e.m(\dots)$	$\text{null}(D)$ si $\text{mtype}(C, m) = \bar{E} \rightarrow D$ et $\vdash e : C$
$(D)e$	$\text{null}(D)$

Cette transformation peut se faire dans n'importe quelle ordre, tant qu'il ne reste plus de nœuds à remplacer à la fin. Il est tout de même préférable de commencer par le nœud le plus proche de la racine afin de procéder au nombre minimal de remplacements.

Nous allons distinguer trois différents cas.

**Premier cas**  $e \rightarrow^* v$  et  $v$  est une valeur.

Alors on note simplement  $V = \{v\}$  (c'était le cas le plus simple).

**Second cas**  $e$  ne se réduit pas en une valeur, et toute suite de réduction partant de  $e$  est finie.

Alors, puisque le nombre d'expressions  $e'$  telles que  $e \rightarrow e'$  est fini pour tout  $e$  et  $e'$ , on peut assurer qu'il y a un nombre fini d'expressions  $e'$  telles que  $e \rightarrow^* e'$ . En appliquant la confluence de la réduction autant de fois qu'il y aie de  $e'$  (moins 1), on peut assurer qu'il existe une expression notée  $e_v$  telle que  $\forall e' \quad e \rightarrow^* e' \implies e' \rightarrow^* e_v$ .

On note enfin notre ensemble  $V = \{\text{varnull}(e_v)\}$ .

**Troisième cas**  $e$  ne se réduit pas en une valeur, et il existe au moins une suite de réductions infinie.

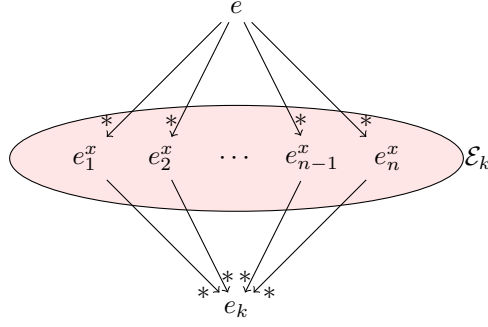
Alors, notons  $\mathcal{E}_k$  l'ensemble des expressions qui sont atteintes depuis  $e$  avec  $k$  réductions ou moins :

$$\mathcal{E}_0 = \{e\} \quad \mathcal{E}_{k+1} = \mathcal{E}_k \cup \rightarrow(\mathcal{E}_k)$$

Par le même argument qu'au paragraphe précédent, les  $\mathcal{E}_k$  sont tous finis.

En appliquant  $\#\mathcal{E}_k - 1$  fois la propriété de confluence, on détermine une expression  $e_k$  (avec  $e_0 = e$ ) telle que

$$\forall e' \quad e' \in \mathcal{E}_k \implies e' \rightarrow^* e_k$$



On note ensuite  $\text{next}(k) = \min\{i \in \mathbb{N}, e_k \in \mathcal{E}_i, i > k\}$

On peut enfin noter la considérer la chaîne infinie (où la puissance dénote la composition).

$$e_{\text{next}^0(0)} \rightarrow^* e_{\text{next}^1(0)} \rightarrow^* e_{\text{next}^2(0)} \rightarrow^* \dots$$

Cette chaîne a la propriété intéressante que l'on puisse y relier n'importe quelle expression  $e'$  issue de  $e$ .

$$\forall e' \quad e \rightarrow^* e' \implies (\exists i \quad e' \rightarrow^* e_{\text{next}^i(0)})$$

En effet, si on note  $i$  le nombre d'étape de  $e \rightarrow^* e'$ , alors  $e' \in \mathcal{E}_i$  donc  $e' \in \mathcal{E}_{\text{next}^i(0)}$  car  $\text{next}$  est strictement croissant et  $\mathcal{E}$  aussi. Donc  $e' \rightarrow^* e_{\text{next}^i(0)}$  par définition de  $e_{\text{next}^i(0)}$ .

Enfin, notre ensemble de valeurs est  $V = \{\text{varnull}(e_{\text{next}^i(0)})\}_{i \in \mathbb{N}}$ .

#### 4.1.3 Preuve du lemme

### 4.2 Exemples montrant l'inutilité de l'équivalence sur les méthodes

```

class IList extends Object {
  IList next;
  Object obj;
  IList(IList next, Object obj) {
    super();
    this.next = next;
    this.obj = obj;
  }
}
class Static extends Object {
  Static() {
    super();
  }
  IList gen(){
 $CT_1 - CT'_1$     return new IList(0, new IList(1, gen()));
 $CT_2 - CT'_2$     return new IList(0, gen());
  }

  Object evilNext(Object x){
    return x.next;
  }
}

```

```

 $TI_1 - TI'_1$  IList{}
 $TI_2 - TI'_2$  IList{Object obj}
 $TI_3 - TI'_3$  IList{IList next, Object obj}
Static ()
Static: IList gen()
 $TI'_1 - TI'_2 - TI'_3$  Static: Object evilNext(Object)

```

On a ici créé comme exemple quatre class tables ( $CT_1, CT'_1, CT_2, CT'_2$ ) et six test interfaces ( $TI_1, TI'_1, TI_2, TI'_2, TI_3, TI'_3$ ) qui vérifient les équivalences suivantes :

	$TI_1$	$TI_2$	$TI_3$
$CT_1/CT_2$	Oui	Non	Non
$CT'_1/CT'_2$	Oui	Oui	Non

On peut montrer les « non » en utilisant les expressions  $e = \mathbf{new}\ Static().gen().next.obj$  et  $e' = \mathbf{new}\ Static().evilNext(\mathbf{new}\ Static().get()).obj$ .

Aussi, dans ces exemples, les class tables  $CT_1/CT_2$  et  $CT'_1/CT'_2$  sont équivalents sous l'angle de toutes les méthodes des test interfaces. Et pourtant ne sont pas équivalentes sous certaines.

On pourrait vouloir palier ce problème en comparant les méthodes sur les valeurs « infinies » qu'elles produisent, mais de la même façon, les class tables ne sont alors jamais équivalentes sous l'angle de `Static.get`, mais elles sont parfois équivalentes (comme sous  $TI_1$ ).

L'exemple  $TI_2$  et  $TI'_2$  nous indique qu'on ne peut pas se contenter de comparer les valeurs de sortie sur les fields laissés apparents par la test interface.

### 4.3 Encore quelques critiques sur la définition de l'équivalence de classes

Je vais encore apporter une critique à la définition de l'équivalence de classes avec les Test Interfaces. Ce ne sont pas ces dernières le problème, mais plutôt la définition du préordre qui en suit. En effet, cette définition « teste » les class table sur toutes les valeurs possibles, et y impose donc des contraintes. Ainsi, deux objets au nom de classes différent seront nécessairement distincts.

En effet, les deux class tables données ?? ne sont pas équivalentes pour la *test interface* donnée (l'expression `new Static().true()`) permet de les distinguer, car `new Bool()` est différent de `new Other()`.

Cependant, on pourrait souhaiter que ces classes soient équivalentes, en effet, le nom de la classe est la seule chose qui permet de les différencier car la classe `Bool` de la première class table est en tout point similaire à la classe `Other` de la seconde ! Un programme qui utilisera la librairie n'aurait jamais à utiliser le nom de la classe du programme et ne saurait distinguer la valeur renvoyée par `Static.true` dans la première class table et dans la seconde.

Afin d'agrandir notre opérateur de préordre  $\prec$ , nous pouvons non plus comparer les *class tables* sur toutes les valeurs, mais peut-être seulement sur celles **constructibles** dans la *test interface*, plus celles **constructibles** par la class table de test.

### 4.4 Renforcement du préordre sur les *class tables*.

On définit d'abord la grammaire des valeurs à trous. On peut la définir comme les expressions à trou, mais en enlevant toutes les règles n'étant pas **new**. On les notera souvent par la lettre  $h$ . Par exemple, `new C([.], new S(new S([.], new G()), [.]))` est une valeur à trois trous. Les valeurs sont bien évidemment identifiées à des valeurs à zéro trous.

On définit alors le préordre sur deux programmes  $(CT_1, e_1) \prec (CT_2, e_2)$  par

$$\forall h \forall \bar{e}' \quad (e_1 \rightarrow_{CT_1}^* h[\bar{e}'] \implies \exists \bar{e}'' \quad e_2 \rightarrow_{CT_2}^* h[\bar{e}''])$$

On peut ainsi redéfinir la comparaison de deux class tables globalement ou sur une méthode :

$$CT_1 \prec CT_2 \iff \forall e \quad (e, CT_1) \prec (e, CT_2)$$

$$CT_1 \prec_{[\text{Class.meth}]} CT_2 \iff \forall \bar{p} \forall \bar{v} \quad (e, CT_1) \prec (e, CT_2) \text{ où } e = \text{new Class}(\bar{p}).\text{meth}(\bar{v})$$

#### Propriété 1

$$\forall h \forall \bar{e} \quad h[\bar{e}] \rightarrow e' \implies \exists \bar{e}'' \quad e' = h[\bar{e}'']$$

*On peut prouver cette propriété inductivement en montrant qu'une expression ayant un nœud « new » en nœud racine se réduira forcément en une expression ayant un nœud racine du même type.*

#### Propriété 2

$$CT_1 \prec_{\text{new}} CT_2 \implies CT_1 \prec_{\text{old}} CT_2$$

*Il s'agit d'une particularisation, puisque qu'une valeur est une valeur à trou sans trou.*

```

class Static extends Object {
  Static() {
    super();
  }
  Bool true(){
    return new Bool();
  }
  Bool false(){
    return new Other();
  }
}
class Bool extends Object {
  Bool() {
    super();
  }
  Object ite(Object t, Object f){
    return return t;
  }
}
class Other extends Bool {
  Other() {
    super();
  }
  Object ite(Object t, Object f){
    return return f;
  }
}

```

```

class Static extends Object {
  Static() {
    super();
  }
  Bool true(){
    return new Other();
  }
  Bool false(){
    return new Bool();
  }
}
class Bool extends Object {
  Bool() {
    super();
  }
  Object ite(Object t, Object f){
    return return f;
  }
}
class Other extends Bool {
  Other() {
    super();
  }
  Object ite(Object t, Object f){
    return return t;
  }
}

```

```

Static()
Static: Bool true()
Static: Bool false()
Bool {}
Bool: Object ite(Object,Object)

```

FIGURE 13 – Deux classes que l'on aimerait équivalentes

**Propriété 3 (Context Lemma)** *L'objectif de cette nouvelle définition était d'obtenir une sorte de context lemma, puisque nous comparons les expressions sur « toutes les valeurs qu'elles pourraient renvoyer ».*

$$\forall CT \forall e_1, e_2 \forall (h, CT_c) \\ (e_1, CT) \prec (e_2, CT) \implies (h[e_1], CT \oplus CT_c) \prec (h[e_2], CT \oplus CT_c)$$

Nous allons le montrer en deux temps. Commençons par décomposer l'opérateur  $\prec$ , et à changer l'ordre des opérateurs :

$$\forall CT \forall e_1, e_2 \forall CT_c \\ (\forall h \quad e_1 \rightarrow^* h[e'] \implies \exists \bar{e}'' \quad e_2 \rightarrow^* h[\bar{e}'']) \implies \\ (\forall h \forall h' \quad h[e_1] \rightarrow^* h'[e'] \implies \exists \bar{e}'' \quad h[e_2] \rightarrow^* h'[\bar{e}''])$$

On va déjà montrer le théorème en ajoutant comme contrainte que la réduction  $h[e_1] \rightarrow^* h'[e']$  s'effectue sans application de méthode (R-METH) à un appel à méthode contenu dans  $h$ . (C'est à dire que les appels à méthode de  $h$  ne « servent à rien », ils pourraient être remplacés par null sans modifier la réduction).

Supposons alors  $H_0 : (\forall h \quad e_1 \rightarrow^* h[e'] \implies \exists \bar{e}'' \quad e_2 \rightarrow^* h[\bar{e}''])$

Montrons ce résultat par **induction** sur  $h$ .

**Premier cas :**  $h = [\cdot]$  Alors on note  $\bar{e}'' = [e_2]$  et donc  $e_2 \rightarrow^* e_2 = h[\bar{e}'']$

**Sinon** alors on a nécessairement  $h = \mathbf{new} \ D(h_1, h_2, \dots, h_k)$ . On note alors  $h[e_1] = \mathbf{new} \ C(h_i[\bar{e}^i])$

$h = [\cdot]$  On applique directement  $H_0$

$h = \mathbf{new} \ D(e_1, \dots, e_{i-1}, h', e_{i+1}, \dots, e'_k)$

Alors,  $D=C$  et  $k' = k$  car  $\mathbf{new} \ D(\dots) \rightarrow^* \mathbf{new} \ C(\dots)$

Aussi, on a pour chaque  $i$ ,  $e_1^i \rightarrow^* h_i[\bar{e}^i]$

Donc  $h[e_2] = \mathbf{new} \ C(e_1^1, \dots, e_{i-1}, h'[e_2], e_{i+1}, \dots, e_1^k)$

Donc  $h[e_2] \rightarrow^* \mathbf{new} \ C(h_1[\bar{e}^1], \dots, h'[e_2], \dots, h_k[\bar{e}^k])$

Or,  $h'[e_1] \rightarrow^* h_i[\bar{e}^i]$ , qui n'utilise pas de réduction de méthode (car sinon, la réduction de  $h[e_1]$  en utiliserai).

Donc, en appliquant l'hypothèse d'induction à  $h'$ , on obtient que  $h'[e_2] \rightarrow^* h_i[\bar{e}^i]$ .

Donc  $h[e_2] \rightarrow^* \mathbf{new} \ C(h_1[\bar{e}^1], \dots, h_i[\bar{e}^i], \dots, h_k[\bar{e}^k]) = h[\bar{e}^i]$

$h = h' . f$

Alors,  $h'[e_1] \rightarrow^* \mathbf{new} \ D(e_1^1, \dots, e_1^{k'})$ , et donc :

$$h[e_1] \rightarrow^* \mathbf{new} \ D(e_1^1, \dots, e_1^{k'}) . f \rightarrow e_1^i \rightarrow^* h[e']$$

$$\text{Donc } h'[e_1] \rightarrow^* \mathbf{new} \ D(e_1^1, \dots, e_1^{k'}) \rightarrow^* \mathbf{new} \ D(e_1^1, \dots, \underbrace{h[e']}_{i\text{ème pos}}, \dots, e_1^{k'}) = h'[e_1^1, \dots, \bar{e}^i, \dots, e_1^{k'}]$$

En appliquant l'hypothèse de récurrence à  $h'$  et  $h'$ , dont la réduction ne contient toujours pas d'appel à méthode (on observe une sous-réduction), on obtient  $h[e_2] = h'[e_2] . f \rightarrow^* h'[\bar{e}^i] . f$

Donc  $h[e_2] \rightarrow^* \bar{h}[\bar{e}']$   
 $\underline{h = h'.meth(\bar{e}) \text{ ou } h = e.meth(\bar{e}, h', \bar{e})}$  Ce cas est impossible, car alors, pour se résoudre en une expression de type **new** (ici,  $\bar{h}[\bar{e}']$ ), il faudrait nécessairement un appel à méthode, ce qui n'est pas présent dans la réduction par hypothèse.

On a donc démontré le lemme pour tous les cas, il nous manque d'inclure les appels à méthode.

**Mélange de réduction** Soient  $h$  et  $\bar{h}$  tels que  $h[e_B] \rightarrow^* \bar{h}[\bar{e}']$  sans autre contrainte. Alors, on peut changer l'ordre des étapes (voire en supprimer) afin d'obtenir une réduction

$$h[e_B] \xrightarrow{\text{R-METH}}^* h'[e_1] \xrightarrow{\underbrace{\quad}^*} \bar{h}[\bar{e}']$$

sans appel à méthode dans  $h'$

## 4.5 Tentative de limitation des class tables

Nous allons dans cette section essayer de prouver le théorème avec plusieurs class tables en les empêchant d'utiliser des fields non présents dans la test interface.

## 4.6 Petites variations dans la class table

Nous allons dans cette section essayer d'utiliser le lemme des arguments finis dans une seule class table pour essayer de la « modifier » en une class table équivalente en modifiant uniquement le corps d'une seule méthode (ou d'un groupe mutuellement récursif).



## **A   Classes et Notations en FeatherweightJava**

### **A.1   La classe Static**

### **A.2   Les booléens**

### **A.3   Les entiers**