

Définition d'un préordre sur les programmes Featherweight Java

Notes sur mon stage au LACL

Samy Avrillon, encadré par
Daniele Varacca (LACL, UPEC)
et Clément Aubert (Augusta University, Géorgie)

22 juillet 2024

Table des matières

1	Introduction	2
1.1	Présentation du problème	2
1.2	Motivations de l'étude	2
1.3	Plan de ce rapport	2
2	Définitions et notations	2
2.1	Rappel sur la définition de Featherweight Java	2
2.2	Définitions et raccourcis de programmation	4
3	Étude d'équivalences simples	5
3.1	Explication de l'utilité d'un contexte	5
3.2	Exemples d'équivalences simples	5
4	Des tests plus ciblés	7
4.1	Définition de la structure de <i>test interface</i>	8
4.2	Définition du typage et d'un premier préordre	10
4.3	Renforcement de ce préordre avec les valeurs infinies	13
4.4	Un exemple concret	15
5	Conclusion	18
6	Bibliographie	19
	Appendices	20

1 Introduction

Ce document a été réalisé dans le cadre de mon stage réalisé au LACL à Créteil, plus de détails sont donnés en [Annexe F](#).

1.1 Présentation du problème

Featherweight Java (abrégié en FJ) [2] est une formalisation de la programmation en orienté objet (POO). Les programmes FJ ont leur syntaxe calquée sur celle du langage Java, et sont compilables avec un compilateur classique. On peut y définir classes, attributs et méthodes, mais pas d'assignation, les objets considérés étant tous immuables.

Le FJ est un bon cas d'étude, si l'on veut étudier formellement un phénomène d'informatique théorique, comme la compilation, le typage incrémental [3], la généricité/polymorphisme ou la possession d'objets, et ce dans un langage orienté objet, paradigme utilisé dans la plupart des langages actuels (Java, Python, C++, ...).

Nous allons utiliser ce langage pour étudier les équivalences entre des programmes, qui n'ont, à notre connaissance, jamais été étudiées en FJ. Trouver une équivalence, cela signifie essayer de dire formellement que deux programmes « font la même chose ». Cela peut être utile afin de vérifier si une optimisation temporelle, mémorielle ou sécuritaire d'un programme ne change pas son fonctionnement.

L'objectif du stage est donc de définir une équivalence (ou plutôt un préordre) entre les programmes FJ et d'essayer d'obtenir des théorèmes utiles sur la nouvelle relation [5].

1.2 Motivations de l'étude

Mes tuteurs de stage, Daniele Varacca et Clément Aubert, s'intéressent aux équivalences de programmes et à comment les *process calculs* peuvent fournir des nouvelles équivalences à d'autres langages [1].

Mon objectif était alors de fournir une réflexion sur une étude de cas (le Featherweight Java), afin d'ensuite pouvoir étudier la manière dont mon exploration « naïve » mais néanmoins approfondie peut-être rapprochée de leur méthode, pouvant ainsi la valider, ou au contraire fournir d'autres éléments afin de la consolider.

1.3 Plan de ce rapport

Je fais [Section 2](#) quelques rappels formels de la définition du Featherweight Java, accompagnés de quelques nouvelles grammaires qui seront utilisées dans ce document. Je vais ensuite présenter [Section 3](#) quelques unes des équivalences, les plus « évidentes », que j'ai considérées afin d'étudier leurs propriétés et d'expliciter leurs problèmes. Ces problèmes, qu'ils soient propres à FJ ou non, devraient être résolus par une « bonne » définition d'équivalence. Enfin, la [Section 4](#) décrira techniquement la relation d'équivalence à laquelle je suis arrivé après toutes ces considérations, et finira par un exemple pratique.

2 Définitions et notations

2.1 Rappel sur la définition de Featherweight Java

Dans cette section, je vais rappeler quelques notations et définitions du langage Featherweight Java [2]. Le lecteur souhaitant toutes les définitions complètes devra lire [2, Section 1].

On utilisera la notation du sur-lignage afin d'indiquer une liste finie d'éléments. Par exemple,

$$f(\bar{a}) \equiv f(a_1, a_2, \dots, a_n)$$

On rappelle la grammaire des expressions, notées e , e_k ou e' :

$e :=$	x	E-VAR
	$ \text{ new } C(\bar{e})$	E-CSTR
	$ e.f$	E-FIELD
	$ e.m(\bar{e})$	E-METH
	$ (C)e$	E-CAST

où x est un nom de variable, C un nom de classe, f un nom d'attribut de classe et m un nom de méthode de classe.

On dit d'une expression qui ne contient pas de référence à des variables ([règle E-VAR](#)) qu'elle est *fermée*. Dans la suite du document, nous allons utiliser le terme « expression » pour désigner les expressions fermées. Les expressions quelconques seront désignées par *expressions ouvertes* ou *expressions à variables*, parfois notées h .

Le langage définit aussi des classes, qui sont composées d'un nombre quelconque d'attributs nommés dont le type est spécifié, d'un unique constructeur qui prend autant de paramètres que la classe a d'attributs et qui définit tous ces derniers, et d'un nombre quelconque de méthodes ayant un nom, un type de retour, des paramètres ayant chacun un nom et un type et surtout un corps, qui est une expression ouverte utilisant les noms de variable décrits dans les paramètres plus éventuellement le nom de variable spécial **this** désignant l'objet duquel la méthode est appelée. Les classes ont aussi une classe mère, par défaut la classe `Object` de laquelle elles héritent les attributs et les méthodes, qu'elles peuvent surcharger sans changer leur type.

On appellera *class table* tout ensemble de définitions de classes, et on les notera \mathcal{A} , \mathcal{B} , \mathcal{C} . Par convention, les noms de classes seront écrit en CamelCase (majuscule) et les noms de méthodes, attributs (*fields*) et variables en camelCase (minuscule).

Un *programme* Featherweight Java est un couple (\mathcal{A}, e) , noté P , Q , R . L'expression est en quelque sorte le *main* du programme.

On appelle « valeur » toute expression composée uniquement de constructeurs ([règle E-CSTR](#)).

On note \vdash la relation de typage. « $\mathcal{A}, \Gamma \vdash e : C$ » indiquera que l'expression e avec l'environnement de typage Γ (un ensemble d'entrées de type $e : C$) est typée par la classe C sous la *class table* \mathcal{A} .

Notez qu'une *class table* n'est pas nécessairement typée. En effet, toutes les classes sont définies individuellement, et c'est le typage de la *class table* complète qui les lie et qui leur donne leur cohérence. Cette propriété nous permet de considérer des *class tables* partielles, qui seront ensuite accolées à d'autres pour être bien typées. FJ définit la notation $\mathcal{A} \text{ OK}$ pour indiquer que la *class table* \mathcal{A} est bien typée. On notera aussi souvent la concaténation de deux *class tables* qui définissent des noms de classe *distincts* $\mathcal{A} \oplus \mathcal{B}$.

La relation de réduction dépend de la *class table* \mathcal{A} utilisée. On la notera $\rightarrow_{\mathcal{A}}$ ou simplement \rightarrow si il n'y a pas d'ambiguïté. On notera aussi $\rightarrow_{\mathcal{A}}^*$ ou \rightarrow^* la clôture transitive et réflexive de la relation. On notera enfin $e \Downarrow v$ lorsque $e \rightarrow^* v$ et v est une valeur. On pourra même écrire $e \Downarrow$ pour dire $\exists v \ e \Downarrow v$.

Cette relation est définie avec deux types de règles [2, Fig. 3], les règles de type R qui indiquent :

- L'évaluation d'un attribut d'un objet (R-FIELD)
- L'évaluation d'une méthode (R-INVK)

```

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

```

FIGURE 1 – Définition de la classe Paire

— L'évaluation d'un *cast* (R-CAST)

Et les règles de type RC qui sont les règles de « congruence », permettant l'évaluation dans n'importe quelle sous-expression.

Enfin, nous reprenons la définition de la classe Paire issue du papier original, reproduite [FIGURE 1](#).

2.2 Définitions et raccourcis de programmation

Nous allons définir en sus la grammaire des expressions à (un seul) trou notés h, h' :

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

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

$$\begin{aligned}
[\cdot][e] &= e \\
(\mathbf{new} \ C(\bar{e}_1, h, \bar{e}_2))[e] &= \mathbf{new} \ C(\bar{e}_1, h[e], \bar{e}_2) \\
(h.f)[e] &= h[e].f \\
(h.m(\bar{e}_1))[e] &= h[e].m(\bar{e}_1) \\
(e_1.m(\bar{e}_2, h, \bar{e}_3))[e] &= e_1.m(\bar{e}_2, h[e], \bar{e}_3) \\
((C)h)[e] &= (C)h[e]
\end{aligned}$$

Nous allons aussi noter $\alpha, \beta, \varepsilon, \gamma$ les *mappings* d'un ensemble de noms de variables vers un ensemble d'expressions fermées. On va ainsi définir la complétion d'une expression à variables par un de ces *mappings*, ce qui sera noté $h[\alpha]$ où l'on remplace chaque occurrence d'une variable par l'image par α de ce nom de variable.

Au niveau de la programmation FJ, nous définissons quelques raccourcis d'écriture. Tout d'abord, la classe `Static` que l'on suppose définie sans attributs. C'est une classe pour laquelle on suppose que les méthodes n'utilisent jamais **this**. On autorise ainsi l'appel des méthodes de cette classe directement, ce qui correspond en pratique à autoriser des expressions de la forme `meth(\bar{e})`, et de les remplacer par `new Static().meth(\bar{e})`.

Enfin nous définissons plusieurs classes qui permettent de coder de façon plus abstraite, dont les définitions complètes sont données en [Annexe A](#).

La première classe définie est la classe `Bool`, liée à deux objets notés **true** et **false**. Cette classe met à disposition une méthode « if then else » `ite(Object, Object)` construite telle que **true**.`ite`(e_t, e_f) renvoie e_t et **false**.`ite`(e_t, e_f) renvoie e_f .

La seconde classe est `Int`, liée à des objets notés 0, 1, 2, Cette classe met à disposition une méthode `Bool isZero()` qui renvoie **true** si l'objet appelant est zéro, et renvoie **false** sinon. Plusieurs méthodes sont définies en sus dans la définition complète présentée en [Annexe A](#) permettant de manipuler ces objets.

Aussi, nous définirons souvent implicitement les classes A, B et C, qui n'ont pas d'attribut et pas de méthode (ce sont des « objets simples »).

3 Étude d'équivalences simples

Dans cette section, je vais présenter les équivalences les plus simples et présenter les problèmes que je leur ai trouvé. Cette section n'a pas pour objectif de construire des théorèmes accompagnés de leurs preuves complètes, mais elle vise à donner au lecteur l'intuition sur les propriétés souhaitables pour un préordre sur les programmes FJ.

3.1 Explication de l'utilité d'un contexte

Les équivalences entre les programmes sont habituellement définies en utilisant des structures appelées « contextes ». Deux programmes sont dits équivalents lorsqu'ils donnent les mêmes résultats sous tout contexte. Pour un contexte C , on va noter $C[P]$ l'interprétation d'un programme P dans le contexte. Alors, le préordre sera défini comme tel :

$$P \prec Q \quad \text{ssi} \quad \forall C \quad C[P] \Downarrow v \implies C[Q] \Downarrow v$$

On confondra souvent les termes « équivalence » et « préordre » dans la suite de ce document, car le premier est plus pratique pour visualiser l'utilité des constructions établies.

Dans des langages formels basés sur une simple expression, comme le λ -calcul ou le π -calcul, cette notion de contexte fonctionne bien. Il suffit de bien la formaliser puis d'étudier ses propriétés [5]. Cependant, en FJ, un programme n'est pas constitué que d'une expression. Une expression seule n'a d'ailleurs aucun sens si elle n'est pas associée à une *class table*. Il faut donc construire un contexte qui tienne compte de l'ensemble du programme.

3.2 Exemples d'équivalences simples

Une définition la plus simple On peut commencer par dire qu'un contexte est simplement une expression à trou $C = h$. L'interprétation d'un programme $P = (\mathcal{A}, e)$ dans un contexte C est alors

$$C[P] = (\mathcal{A}, h[e])$$

Le *premier* problème de cette définition est qu'elle est très peu puissante. En effet, le contexte est limité à l'utilisation de la *class table* testée. Un exemple de deux *class tables* qui ne peuvent être différenciées que à l'aide d'une *class table* supplémentaire « de tests » est donnée [Annexe B](#).

Le *second* problème est que les expressions peuvent observer de manière trop précise et profonde la *class table* est ainsi les distinguer au moindre changement de nom de classe, de nom d'attribut ou de nom de méthode. On peut s'en convaincre en considérant les *class tables* qui contiennent des trous dégénérés, c'est à dire celles vérifiant

$$\forall e \exists h \forall p \quad h[p] \rightarrow^* e$$

Un exemple de trou dégénéré est l'expression trouée suivante donnée pour e expression, dans n'importe quelle *class table* contenant la classe Paire :

$$\mathbf{new} \text{ Paire}([\cdot], e) . \text{snd} \rightarrow e$$

Avec un trou dégénéré, on peut alors « jeter » le *main* du programme, et comparer les deux *class tables* sur n'importe quelle expression. Le *main* du programme n'a plus d'effet. On peut donc évaluer pour chaque nom de classe C l'expression $\mathbf{new} \ C(\mathbf{new} \ D(\dots), \dots, \mathbf{new} \ E(\dots))$ qui est une valeur si et seulement si la classe C est présente dans la *class table* et que ses attributs sont de types D, \dots, E (et que ces types sont instanciables, c'est à dire qu'il existe une valeur de ce type). De la même manière, on peut tester si une méthode existe dans une classe (à supposer qu'il existe des paramètres telle qu'elle se réduise en valeur). Donc deux programmes comparés ainsi devront avoir exactement la même structure, les mêmes noms de classe, mêmes noms d'attributs, même noms de méthode.

Ainsi, on ne peut pas comparer par exemple une implémentation d'un algorithme de tri qui utiliserait une classe Tree et une autre qui utiliserait une classe Map. Puisqu'il n'y a pas d'*access control* en FJ, un utilisateur malveillant aurait accès à tous ces outils pour mettre en défaut le programme.

La relation créée en utilisant de simples expressions à trou comme contextes est ainsi robuste, mais trop restrictive pour être utile.

Une boîte à outils plus puissante Afin de permettre aux contextes d'être plus complets et d'ainsi résoudre le premier problème, on est d'abord tenté d'y accoler une *class table* supplémentaire dite «de tests». Ainsi, l'application du contexte $C = (\mathcal{C}, h)$ au programme $P = (\mathcal{A}, e_P)$ donne

$$C[P] = (\mathcal{C} \oplus \mathcal{A}, h[e_P])$$

(le \oplus dénote un α -renommage évitant que les définitions s'écrasent les unes les autres).

Hélas, cela renforce le second problème, car il existe alors forcément un trou dégénéré dans une *class table* $\mathcal{C} \oplus CT_P$ (on n'a qu'à rajouter une classe ressemblant à Paire).

Restreindre les *class tables* Une dernière définition simple consiste à imposer certaines contraintes sur les *class tables* des programmes comparés. On va d'abord définir le préordre suivant sur les *class tables* :

$$\mathcal{A} \prec \mathcal{B} \iff \forall e \forall v \quad (\mathcal{A}, e) \Downarrow v \implies (\mathcal{B}, e) \Downarrow v$$

Souhaitant garder les contextes les plus puissants possibles, on les définit comme des couples (\mathcal{C}, h) et on note alors la contextualisation d'un programme $P = (\mathcal{A}, e)$

$$C[P] = (\mathcal{C}, h[e]) \quad \text{si } \mathcal{A} \prec \mathcal{C}$$

Cette condition restreint le nombre de contextes pouvant transformer un programme, et donc transforme le préordre en

$$P = (\mathcal{A}, e_P) \prec Q = (\mathcal{B}, e_Q) \iff \forall C = (\mathcal{C}, e_C) \quad \begin{cases} \mathcal{A} \prec \mathcal{C} \\ \mathcal{B} \prec \mathcal{C} \\ \forall v \quad C[P] \Downarrow v \implies C[Q] \Downarrow v \end{cases}$$

Cependant, cette définition pose un problème lorsque l'on cherche à comparer deux *class tables* qui n'ont aucune « sur-*class table* » qui convienne aux deux en même temps. C'est le cas des deux classes données [FIGURE 2](#) notées \mathcal{A} et \mathcal{B} .

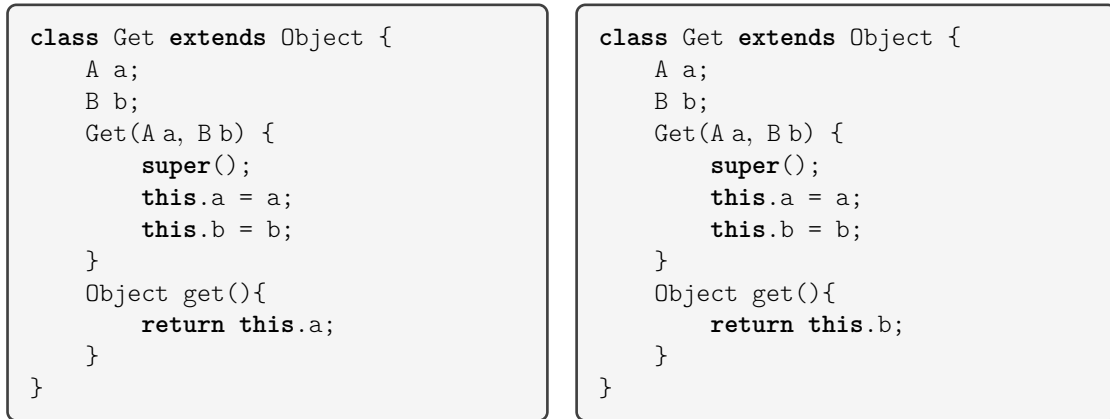


FIGURE 2 – *Class tables* n’ayant pas de sur-*class table* commune

En effet, en notant les expressions suivantes

$$\begin{aligned}
e_A &= (A) (\text{new Get}(\text{new A}(), \text{new B}()).\text{get}()) \\
e_B &= (B) (\text{new Get}(\text{new A}(), \text{new B}()).\text{get}())
\end{aligned}$$

On peut observer les relations suivantes qui montrent que une *class table* qui conviendrait à A et à B serait impossible, car elle pourrait et ne pourrait pas évaluer e_A et e_B en une valeur.

$$\begin{aligned}
(CL_A, e_A) \Downarrow \text{new A}() & \quad (CL_A, e_B) \not\Downarrow \\
(CL_B, e_A) \not\Downarrow & \quad (CL_B, e_B) \Downarrow \text{new B}()
\end{aligned}$$

Alors, puisque aucun de nos *contextes* ne satisfait les hypothèses, les deux programmes sont équivalents trivialement.

4 Des tests plus ciblés

Abandon de l’expression main Tous les tests présentés dans la section précédente visaient à tester un programme entier (une *class table* et une expression). Il semble cependant étrange de procéder ainsi. En effet, un développeur souhaitera principalement comparer entre elles des bibliothèques, ce qui correspondrait à nos *class tables*. On ne cherche plus à savoir si deux programmes « font la même chose », mais plutôt est-ce que ces deux *class tables* fournissent les mêmes fonctions.

En bonus, l’étude des *class tables* est tout aussi complète que l’étude des programmes entiers, car nous pouvons toujours rajouter une classe Main qui n’aie qu’une méthode Main.main dont le corps soit l’expression de notre programme. Alors tester cette nouvelle *class table* reviendra à tester le programme entier. La transformation est présentée [FIGURE 3](#).

Alors, on a transformé le *programme* en *class table*, et si nous voulons tester le main du programme, il suffit de tester ce que renvoie l’expression `new Main().main()`, qui se réduit en une étape en le main du programme.

Idée d’une nouvelle structure Le problème de tous les contextes ci-dessus est qu’ils imposent tous une structure trop précise aux *class tables* comparées. Elles doivent toutes avoir impérati-

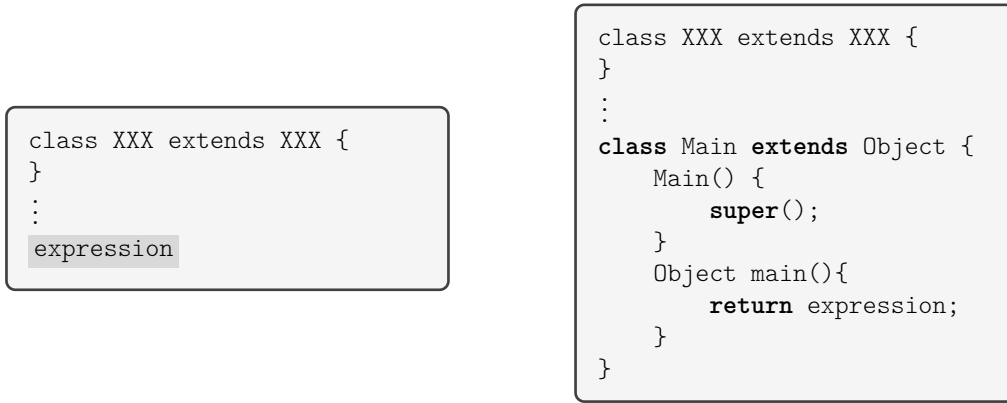


FIGURE 3 – Transformation d'un programme en *class table*

TR	:=	C : C m(\overline{C})	TIG-METH
		C { \overline{C} \overline{f} }	TIG-ATTR
		C (\overline{C} $\overline{?f}$)	TIG-CSTR
		C <: C	TIG-CAST

$\overline{?f}$ désigne une liste d'éléments qui sont ou bien vides, ou bien un nom d'attribut, par exemple $f1, vide, vide, f2$

FIGURE 4 – Grammaire des *Test Interface rules*

vement les mêmes classes, attributs et méthodes, là où on aimerait justement les comparer sur certaines classes, attributs ou méthodes.

Nous allons donc créer une structure appelée *test interface* (ou interface de test) qui permet de restreindre les tests à l'utilisation de classes, attributs et méthodes spécifiques. La restriction se fera à l'aide du typage.

Le terme interface n'est pas anodin, la structure ressemble à une interface Java (que certaines extensions du FJ considèrent [4]), sauf que les *test interface* ne serviront que lors de la compilation. Une expression va être « compilée » (c'est à dire être typée) avec l'interface, puis sera « exécutée » (c'est à dire réduite) avec des vraies *class tables*.

4.1 Définition de la structure de *test interface*

Structure Nous définissons donc la structure des *test interface rules* dans la FIGURE 4. Une *test interface* est alors simplement un ensemble de *test interface rules*, et sera notée $\mathfrak{T}, \mathfrak{U}, \mathfrak{V}$. La règle TIG-METH permet de déclarer une méthode dans une classe et de spécifier son type. La règle TIG-ATTR permet de déclarer certains attributs d'une classe, en spécifiant leur nom et leur type. La règle TIG-CSTR permet de déclarer le constructeur de la classe ainsi que les types des attributs, et éventuellement certains noms. Enfin, la règle TIG-CAST permet d'indiquer qu'une classe doit être un sous-type d'une autre. Et donc, une *class table* implémente une *test interface* lorsqu'elle respecte chacune de ses règles.

Un exemple concret de *test interface* est donné FIGURE 5. Elle impose la déclaration des


```

Number {}

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

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

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

```

FIGURE 5 – Exemple de *test interface*

classes `Int`, `Frac`, `Number` et `RichInt` telles que l'on puisse appeler certaines méthodes sur elles. Elle impose aussi de pouvoir construire les object `Frac` avec deux paramètres de type `Int`, et elle impose aussi un attribut nommé `value` dans la classe `RichInt`.

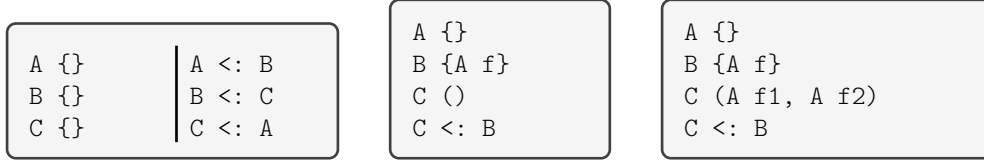
À cette grammaire, nous allons ajouter quelques règles afin que la *test interface* soit considérée comme valide. Dans la suite du document, on considèrera que toutes les *test interfaces* utilisées sont toujours valides. Ces règles sont les suivantes :

- Chaque nom de classe est défini au plus une fois par ou bien une [règle TIG-ATTR](#) ou bien une [règle TIG-CSTR](#).
- Chaque nom de methode est défini au plus une fois par nom de classe par une [règle TIG-METH](#).
- Les noms de classe utilisés sont définis (sauf éventuellement `Object`).

Cette troisième règle permet d'éviter les définitions implicites, en suivant la convention du papier original [2]. On peut dans tous les cas rajouter pour une classe `C` non définie une ligne de la forme `C {}`.

Les deux règles [TIG-ATTR](#) et [TIG-CSTR](#) sont donc mutuellement exclusives pour un nom de classe donné. Leur différence est que la seconde autorise le développeur à appeler le constructeur `new C(...)` et impose la position de chaque attribut déclaré dans les paramètres du constructeur. Il impose aussi le nombre d'attributs de la classe (en prenant en compte l'hérédité). La première n'autorise pas l'utilisation du constructeur, et n'impose donc pas d'ordre sur ses paramètres.

Il peut sembler au premier abord qu'il manque à cette définition des *test interfaces* une façon d'autoriser le programmeur à utiliser un constructeur, sans en imposer l'ordre des paramètres, mais en autorisant l'accès à certains attributs. Pourtant, si l'on peut construire et accéder à un certain attribut `field` sur un type `C`, et que l'objet est constructible, c'est à dire qu'il existe une expression `new C(new P1(e1), ..., new Pn(en))`, alors on peut créer une expression qui nous indique la position dans le constructeur de l'attribut. Pour ce faire, il faut, pour chaque nom de classe `Pk` des paramètres, créer une sous-classe `Pk_k` (les `Pk` pouvaient se confondre, mais les `Pk_k` sont tous distincts). Alors la construction suivante renverra une valeur de type `Pj_k`, et on



La première parce que la relation de sous-typage et transitive est antisymétrique.

La seconde parce que si B a un attribut de type A, alors C en hérite, et ne peut pas être construit sans attributs.

La troisième comme la précédente, parce que C devrait avoir par héritage un attribut nommé f.

FIGURE 6 – Exemples de *test interfaces* implémentés par aucune *class table*

aura ainsi accès à la position dans le constructeur de l'attribut field.

```
new C(new P1_1(e1), new P2_2(e2), ..., new Pn_n(en)).field
```

Nous allons aussi considérer uniquement les *test interfaces* pour lesquelles il existe au moins une *class table* les implémentant. Cela impose certaines règles, notamment que l'opération $<:$ induite puisse créer un bon ordre, mais d'autres contraintes plus complexes à exprimer apparaissent, nous n'allons pas chercher à les exprimer ici, mais vous trouverez des contre-exemples de *test interfaces* en [FIGURE 6](#)

Opérateur d'implémentation Nous allons maintenant définir un opérateur « d'implémentation » noté \triangleright , qui décrit l'idée qu'une *class table* \mathcal{A} implémente une *test interface*, c'est à dire qu'elle vérifie toutes les contraintes que cette dernière impose. Les règles d'inférence définissant la relation sur les *test interface rules* sont définies [FIGURE 7](#). On indique ensuite que $\mathcal{A} \triangleright \mathfrak{T}$ lorsque pour toute règle $TR \in \mathfrak{T}$, $\mathcal{A} \triangleright TR$, et que \mathcal{A} est bien typé ($\mathcal{A} \text{ OK}$).

L'implémentation a été choisie la plus large possible, par exemple, une méthode $C.m$ typée par $\text{Object} \rightarrow \text{Number}$ implémentera une règle $[C : \text{Int } m(A)]$. En effet, cette construction n'a pas pour objectif d'étudier le *typage* de Featherweight Java, mais bien son *fonctionnement*.

4.2 Définition du typage et d'un premier préordre

Maintenant, nous allons définir le typage d'expressions dans une *test interface*. L'opérateur de typage sera noté \Vdash pour le différencier de celui défini dans FJ [\[2, Fig.2\]](#), notée \vdash . Une expression sera typée sous une *test interface* \mathfrak{T} , une *class table* \mathcal{B} et un environnement de typage Γ .

Pour simplifier la définition du typage, nous allons (re)définir les applications suivantes :

fields renvoie l'ensemble des attributs de la classe spécifiée.

mtype renvoie le type de la méthode spécifiée dans la classe spécifiée.

mbody renvoie le corps de la méthode spécifiée dans la classe spécifiée (donc une expression avec variables).

construct renvoie le type du constructeur de la classe spécifiée, c'est à dire la liste finie des types des paramètres.

Ces applications étaient déjà définies dans le papier original [\[2, Fig.1\]](#) et « recherchaient » dans une simple *class table*, notre redéfinition les définit recherchant dans une *class table* mais aussi dans une *test interface*. Les définitions étendues sont présentées [FIGURE 8](#).

$\frac{\text{mtype}_{\mathcal{A}}(\text{m}, \text{C}) = \bar{\text{E}} \rightarrow \text{H} \quad \bar{\text{F}} <_{:\mathcal{A}} \bar{\text{E}} \quad \text{H} <_{:\mathcal{A}} \text{G}}{\mathcal{A} \triangleright [\text{C} : \text{G} \text{ m}(\bar{\text{F}})]}$	(TRI-METH)
$\frac{\bar{\text{F}} \bar{\text{f}} \subset \text{fields}_{\mathcal{A}}(\text{C}) \quad \bar{\text{F}} <_{:\mathcal{A}} \bar{\text{E}}}{\mathcal{A} \triangleright [\text{C} \{ \bar{\text{E}} \bar{\text{f}} \}]}$	(TRI-ATTR)
$\frac{\begin{array}{l} \exists \bar{\text{f}}' \quad \bar{\text{f}}\bar{\text{O}} = \bar{\text{f}} \boxplus \bar{\text{f}}' \\ \bar{\text{F}} = \bar{\text{E}} \quad \text{où } \bar{\text{f}} \text{ est défini} \\ \bar{\text{E}} <_{:\mathcal{A}} \bar{\text{F}} \quad \wedge \quad \bar{\text{F}} \bar{\text{f}}\bar{\text{O}} = \text{fields}_{\mathcal{A}}(\text{C}) \end{array}}{\mathcal{A} \triangleright [\text{C} \langle \bar{\text{E}} \bar{\text{f}} \rangle]}$	(TRI-CSTR)
$\frac{\text{C} <_{:\mathcal{A}} \text{D}}{\mathcal{A} \triangleright [\text{C} <_{:\mathcal{A}} \text{D}]}$	(TRI-CAST)

$\text{mtype}_{\mathcal{A}}$ est une application qui renvoie le type de la méthode spécifiée de la classe spécifiée [2, Fig.1], dans la *class table* \mathcal{A} .

$\text{fields}_{\mathcal{A}}$ est une application qui renvoie la liste complète des attributs de la classe spécifiée [2, Fig.1], dans la *class table* \mathcal{A} .

$\bar{\text{f}} \boxplus \bar{\text{f}}'$ désigne la liste $\bar{\text{f}}$ dont *tous* les attributs manquants ont été complétés par les éléments de la liste $\bar{\text{f}}'$.

FIGURE 7 – Définition de l'opérateur *Test interface Rule Implemented* (règles TRI)

Grâce à ces méthodes, nous pouvons réutiliser les règles de typage de Featherweight Java presque à l'identique. Nous les donnons FIGURE 9. Nous avons enlevé la règle de typage T-SCAST qui autorisait les casts que les auteurs qualifiaient de « stupides », entre deux types qui ne sont pas parents l'un de l'autre.

Nous vérifions aussi évidemment la *class table* \mathcal{B} , en vérifiant que le corps de chaque méthode est bien typé par le type attendu dans la méthode, que les champs de la *class table* ne contredisent pas la *test interface*. On note alors $\mathcal{B} \text{ OK IN } \mathfrak{T}$.

Nous allons maintenant vérifier la définition de cette opération de typage \Vdash en démontrant le théorème suivant :

Théorème 1 *Soit une test interface \mathfrak{T} , une class table \mathcal{A} , un couple (\mathcal{B}, e) class table \times expression fermée appelée « test », et un environnement de typage Γ vérifiant*

- $\mathcal{A} \triangleright \mathfrak{T}$
- $\mathcal{B} \text{ OK IN } \mathfrak{T}$
- $\mathfrak{T}, \mathcal{B}, \Gamma \Vdash e : \text{D}$

Alors il existe C tel que $\mathcal{A} \oplus \mathcal{B}, \Gamma \vdash e : \text{C}$ et $\text{C} < \text{D}$.

Cette dernière condition est nécessaire à cause de la tolérance de l'opérateur \triangleright .

Ce théorème se prouve par induction sur la preuve de $\mathfrak{T}, \mathcal{B} \Vdash e : \text{D}$. La preuve est donnée en Annexe C.

On peut alors définir pour chaque *test interface* un préordre sur les *class tables* qui l'implémentent de la manière suivante.

$\frac{[C(\bar{D} \bar{f})] \in \mathfrak{T}}{\text{construct}(C) = \bar{D}}$	(LT _I -CSTR)
$\frac{\text{class } C\{C(\bar{D} \bar{f})\{\dots\} \dots\} \in \mathcal{B}}{\text{construct}(C) = \bar{D}}$	(LCT-CSTR)
$\frac{[C \ \{\bar{D} \bar{f}\}] \in \mathfrak{T}}{\text{fields}(C) = \bar{D} \bar{f} + \bigcup_{C <: E} \text{fields}(E)}$	(LT _I -ATTR)
$\frac{[C \ (\bar{D} \bar{f})] \in \mathfrak{T}}{\text{fields}(C) = \underbrace{\bar{D} \bar{f}}_{\text{pour } \bar{f} \text{ défini}} + \bigcup_{C <: E} \text{fields}(E)}$	(LT _I -ATTRC)
$\frac{\text{class } C \ \{\bar{D} \bar{f}; \dots\} \in \mathcal{B}}{\text{fields}(C) = \bar{D} \bar{f} + \bigcup_{C <: E} \text{fields}(E)}$	(LCT-ATTR)
$\frac{[C : E \ m(\bar{D})] \in \mathfrak{T}}{\text{mtype}(m, C) = \bar{D} \rightarrow E}$	(LT _I -METHC)
$\frac{\text{class } C \ \{\dots; E \ m(\bar{D} \bar{x})\} \in \mathcal{B}}{\text{mtype}(m, C) = \bar{D} \rightarrow E}$	(LCT-METH)
$\frac{\text{mtype}(m, C') = \bar{D} \rightarrow E \quad C <: C'}{\text{mtype}(m, C) = \bar{D} \rightarrow E}$	(LUP-METH)

$<$: dénote ici la cloture transitive et réflexive de l'union des relations $<$: sur \mathfrak{T} et *extends* dans \mathcal{B} .

FIGURE 8 – Opérateurs supplémentaires pour le typage

$\frac{x \in \Gamma}{\mathfrak{T}, \mathcal{B}, \Gamma \Vdash x : \Gamma(x)}$	(TI-VAR)
$\frac{\mathfrak{T}, \mathcal{B}, \Gamma \Vdash e : Cc \quad Cc.f \in \text{fields}(Cc)}{\mathfrak{T}, \mathcal{B}, \Gamma \Vdash e.f : C}$	(TI-FIELD)
$\frac{\mathfrak{T}, \mathcal{B}, \Gamma \Vdash e : Cc \quad \mathfrak{T}, \mathcal{B}, \Gamma \Vdash \overline{e_x} : \overline{CX} \quad \overline{CX} <: \overline{D} \quad \text{mtype}(m, Cc) = \overline{D} \rightarrow C}{\mathfrak{T}, \mathcal{B}, \Gamma \Vdash e.m(\overline{e_x}) : C}$	(TI-INVK)
$\frac{\mathfrak{T}, \mathcal{B}, \Gamma \Vdash \overline{e_x} : \overline{CX} \quad \overline{CX} <: \overline{D} \quad \text{constructor}(C) = \overline{D}}{\mathfrak{T}, \mathcal{B}, \Gamma \Vdash \mathbf{new} \ C(\overline{e_x}) : C}$	(TI-NEW)
$\frac{\mathfrak{T}, \mathcal{B}, \Gamma \Vdash e : D \quad C <: D \quad C \neq D}{\mathfrak{T}, \mathcal{B}, \Gamma \Vdash (C)e : C}$	(TI-UCAST)
$\frac{\mathfrak{T}, \mathcal{B}, \Gamma \Vdash e : D \quad D <: C}{\mathfrak{T}, \mathcal{B}, \Gamma \Vdash (C)e : C}$	(TI-DCAST)

FIGURE 9 – Typage des expressions avec une *test interface* et une *class table*

Definition 1 Soit une *test interface* \mathfrak{T} .

Soient deux *class tables* \mathcal{A} et \mathcal{B} qui implémentent toutes deux \mathfrak{T}

Alors

$$\mathcal{A} \prec_{\mathfrak{T}} \mathcal{A}' \quad \text{ssi} \quad \forall (\mathcal{B}, e) \left| \begin{array}{l} \mathcal{B} \text{ OK IN } \mathfrak{T} \\ \mathfrak{T}, \mathcal{B} \Vdash e : D \\ \mathcal{A} \oplus \mathcal{B} \Downarrow v \end{array} \right. \implies \mathcal{A}' \oplus \mathcal{B} \Downarrow v$$

où $\mathcal{A} \oplus \mathcal{B}$ dénote l'ensemble des classes de \mathcal{A} auquel on a ajouté les classes de \mathcal{B} dont les noms n'étaient pas déjà dans \mathcal{A} .

Nous noterons la plupart du temps simplement \prec pour désigner $\prec_{\mathfrak{T}}$ si la *test interface* considérée est sans ambiguïté.

Nous allons déjà renforcer ce préordre, avant de fournir un exemple complet [Sous-section 4.4](#).

4.3 Renforcement de ce préordre avec les valeurs infinies

Nous souhaiterions pouvoir démontrer un *context lemma* sur les préordres précédents, c'est à dire un théorème qui dise que pour toute expression à trou h

$$(\mathcal{A}, e) \prec (\mathcal{B}, f) \implies (\mathcal{A}, h[e]) \prec (\mathcal{A}, h[f])$$

Cependant, cela est généralement faux, puisque il est possible en Featherweight Java de considérer des expressions qui acceptent une suite infinie de réductions, mais que l'on peut « utiliser » avec des appels à méthodes ou des attributs. Nous pouvons par exemples utiliser les listes infinies, définies en annexe ([Sous-section A.3](#)). Cette classe `IList` définit deux attributs, dont un de son propre type.

```

class Static extends Object {
  Static() {
    super();
  }
  IList intList(Int i){
    return new IList(i, intList(i+1));
  }
  IList zeroList(){
    return new IList(0, zeroList());
  }
}

```

FIGURE 10 – Définitions de listes infinies

Par exemple, avec la *class table* définie [FIGURE 10](#), nous pouvons considérer les deux expressions $e_i = \text{intList}(0)$ et $e_0 = \text{zeroList}()$. Tels que nous les avons définis précédemment, nos préordres ne permettent pas de différencier ces deux expressions. En effet, puisque aucune des deux ne se réduit en une valeur, on est dans une situation où les prémisses ne sont jamais vérifiées, et donc, l'universalité du vide s'applique. Cependant, certaines expressions à trou sont capables de les différencier. On peut par exemple considérer l'expression $h = [\cdot].\text{next}.\text{obj}$, nous aurons effectivement les réductions suivantes :

$$\begin{aligned}
h[e_i] &= \text{intList}(0).\text{next}.\text{obj} \\
&\rightarrow \text{new IList}(0, \text{intList}(0+1)).\text{next}.\text{obj} \\
&\rightarrow \text{intList}(0+1).\text{obj} \\
&\rightarrow \text{new IList}(0+1, \text{intList}(0+1+1)).\text{obj} \\
&\rightarrow 0+1 \\
&\rightarrow^* 1 \\
h[e_0] &= \text{zeroList}().\text{next}.\text{obj} \\
&\rightarrow \text{new IList}(0, \text{zeroList}()).\text{next}.\text{obj} \\
&\rightarrow \text{zeroList}().\text{obj} \\
&\rightarrow \text{new IList}(0, \text{zeroList}()).\text{obj} \\
&\rightarrow 0
\end{aligned}$$

Nous avons l'impression que certaines expressions forment des sortes de « valeurs infinies », qui peuvent quand même être utilisées dans des programmes, mais uniquement de manière finie. Nous allons donc changer la définition de tous nos préordres, en remplaçant les propositions de la forme suivante :

$$\forall v \quad (\mathcal{A}, e) \Downarrow v \implies (\mathcal{B}, f) \Downarrow v$$

Nous allons tout d'abord définir une nouvelle grammaire, celle des « valeurs ouvertes » ou « valeurs avec variables », que l'on notera avec la lettre \tilde{h} . Il s'agit simplement d'expressions ouvertes qui ne contiennent que des nœuds de type [E-VAR](#) et [E-CSTR](#) et telles que un nom de variable n'est utilisé au plus qu'une seule fois.

Par exemple, l'objet suivant est une valeur à trois variables

```
new C(x, new S(new S(y, new G()), z))
```

On a donc aussi une opération de remplissage, comme pour les expressions à variables, que l'on note de la même manière. On supposera que si l'on marque $\tilde{h}[\alpha]$, c'est que l'ensemble des variables libres de \tilde{h} est inclus dans l'ensemble de définition de α .

Nous allons alors pouvoir redéfinir le préordre sur deux programmes :

$$(\mathcal{A}, e) \preceq (\mathcal{B}, f) \quad \text{ssi} \quad \forall h \forall \alpha \quad (e \rightarrow_{\mathcal{A}}^* h[\alpha] \implies \exists \beta \quad f \rightarrow_{\mathcal{B}}^* h[\beta])$$

Intuitivement, on vérifie que si d'un côté du préordre, l'expression peut s'évaluer en un objet de type C, alors l'autre côté s'évaluera lui aussi en un objet de type C, et ce récursivement.

On change ainsi dans la définition précédente du préordre sur les *class tables*, mais aussi dans tout autre préordre « simple » que l'on aurait pu définir :

$$\mathcal{A} \preceq \mathcal{B} \quad \text{ssi} \quad \forall e \quad (\mathcal{A}, e) \preceq (\mathcal{B}, e)$$

Nous allons maintenant étudier les propriétés de cette nouvelle définition du préordre.

Propriété 1

$$\forall h \forall \alpha \quad h[\alpha] \rightarrow^* e' \implies \exists \beta \quad e' = h[\beta]$$

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, en inversant la relation de réduction. La preuve complète est présentée [Annexe D](#).

Propriété 2

$$\mathcal{A} \preceq \mathcal{B} \implies \mathcal{A} \prec \mathcal{B}$$

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

Propriété 3 (Context Lemma)

$$\begin{aligned} & \forall \mathcal{A} \quad \forall e, f \quad \forall (h, \mathcal{B}) \\ (e, \mathcal{A}) \preceq (f, \mathcal{B}) & \implies (h[e], \mathcal{A} \oplus \mathcal{B}) \preceq (h[f], \mathcal{A} \oplus \mathcal{B}) \end{aligned}$$

Ce théorème était en quelque sorte l'objectif de cette redéfinition, puisque nous comparons dorénavant les expressions sur « toutes les valeurs qu'elles pourraient renvoyer ».

La démonstration se fera sur une version plus puissante du théorème où h peut prendre plusieurs trous. La preuve se fait ensuite par récurrence sur la longueur de la chaîne de réduction, en discriminant selon l'emplacement de la réduction (dans h , dans e ou entre les deux), et en changeant les noms de variables afin de pouvoir boucler la récurrence. La démonstration complète est présentée en [Annexe E](#).

4.4 Un exemple concret

Je vais enfin présenter un exemple de *class tables* qui sont différentes et qui seront équivalentes sous une certaine *test interface* que je donnerai également.

Ces *class tables* auront pour objectif de définir une fonction de tri de listes (`Sort.sort`), en utilisant deux algorithmes récursif différents. Le premier qui trouve le minimum de la liste, le place au début de cette dernière puis trie le reste. Le second trouve le maximum de la liste, le place à la fin (à l'aide d'un accumulateur), puis trie le reste.

La première *class table* [FIGURE 11](#) et la seconde *class table* [FIGURE 12](#) définissent toutes deux une première classe d'utilité afin de calculer des minimums/maximums. Les secondes classes sont celles qui font le processus de tri à proprement parler.

Enfin, la *test interface* présentée [FIGURE 13](#) est assez restreinte afin d'éviter de trop discriminer nos deux classes.

```

class MinGetter extends Object {
  /*
    Renvoie le plus petit des entiers donnés.
  */
  Int min(Int a, Int b){
    return a.lt(b).ite(a,b);
  }
  /*
    Renvoie le minimum de la liste, renvoie None si elle est vide
    .
  */
  Optional listMin(List a) {
    return a.ensureFinite().isEmpty().ite(new None(), new Some(
      this.min((Int)this.listMin(a.tail).orElse(a.head), a.head)
    ))
  }
}
class Sort extends Object {
  List sort(List in){
    ensure(in).isEmpty().ite(in,minOnStart(in,(Int)((Some)new
      MinGetter().listMin(in)).value));
  }
  /*
    Place la première occurrence de 'min' trouvée dans 'in' au dé
    but de la liste, et trie le reste.
  */
  List minOnStart(List in, Int min) {
    return min::this.sort(in.removeFirst(min));
  }
}

```

FIGURE 11 – Première *class tables*


```

class MaxGetter extends Object {
  /*
    Renvoie le plus grand des entiers donnés.
  */
  Int max(Int a, Int b){
    return a.gt(b).ite(a,b);
  }
  /*
    Renvoie le maximum de la liste, renvoie None si elle est vide
    .
  */
  Optional listMax(List a) {
    return a.ensureFinite().isEmpty().ite(
      new None(),
      new Some(this.max((Int)this.listMax(a.tail).orElse(a.head
        ), (Int)a.head)))
  }
}

class Sort extends Object {

  List sort(List in){
    return this.sortAcc(in, []);
  }

  List sortAcc(List in, List acc) {
    ensure(in).isEmpty().ite(acc,maxToEnd(in,((Some)new MaxGetter
      ().listMax(in)).value));
  }

  /*
    Enlève 'max' de la liste 'in', et applique sortAcc en
    ajoutant max à l'accumulateur.
  */
  List maxToEnd(List in, Int max) {
    return this.sortAcc(in.removeFirst(max), max::in.removeFirst(
      max));
  }
}

```

FIGURE 12 – Seconde *class tables*

```

// Ces trois lignes permettent de construire les entiers
Int ()
S (Int)
S <: Int

// Ces trois lignes permettent de construire les listes
List ()
Cons (List, o)
Cons <: List

// Ces deux lignes permettent l'utilisation des méthodes de tri
// C'est le seul endroit où les deux class tables diffèrent du point
// de vue de la test interface.
Sort ()
Sort: sort(List)

```

FIGURE 13 – La *test interface* utilisée pour la comparaison

Je ne vais pas présenter ici une preuve formelle de l'équivalence de ces deux classes, notamment parce que c'est assez compliqué sans d'autres théorèmes, la définition étant sémantique, il nous manquerait un théorème qui permettrait de se ramener, par exemple, à l'évaluation de chaque méthode. J'ai quelque peu essayé de construire certains de ces théorèmes, mais je n'ai pas eu le temps d'en obtenir un utile avant la fin de mon stage (même si le *context lemma* fournit un très bon moyen de prouver de tels théorèmes).

Algorithmiquement, nous pouvons nous convaincre que les méthodes font ce qui leur est demandé, et l'équivalence algorithmique des deux façons de faire est assez simple.

Cet exemple nous fournit néanmoins quelques remarques supplémentaires. Nous pouvons remarquer que certaines constructions ont dû être utilisées afin de s'assurer qu'un utilisateur ne n'ait pas intercalé des classes différentes altérant le fonctionnement de notre programme, notamment pour la classe *List* qui est utilisable par le programme de test. Une solution qui pourrait être envisagée (et d'ailleurs celle retenue en Java classique) est l'ajout d'un mot-clé *final* sur les noms de classes des *test interfaces* qui empêcherait aux classes d'une *class table* l'implémentant d'étendre d'une classe marquée *final*, au delà des extensions indiquées.

5 Conclusion

Nous avons donc créé un préordre en définissant une nouvelle structure de *test interface*. Cette structure est à la fois assez puissante pour que l'on puisse tester profondément les programmes, mais aussi assez tolérante pour que l'équivalence ne s'arrête pas sur la moindre différence de grammaire. Nous avons aussi obtenu un *context lemma* pour notre préordre, ce qui nous conforte dans son bon fondement.

Nous pourrions complexifier la structure en observant d'autres problèmes, par exemple, la comparaison des valeurs de retour (finies ou infinies) n'a pas de limitation sur les types des objets. Nous pourrions imaginer une librairie dont on aimerait qu'elle renvoie un objet *State* décrivant son état, que la *test interface* n'autorise pas à utiliser, si ce n'est comme argument d'une méthode. Alors, la comparaison telle qu'elle est faite va vérifier les variables « internes »

de State alors que nous aimerions laisser l'implémentation libre.

La preuve complète des théorèmes a pris plus de temps que prévu, notamment à cause de la structure moins intuitive de la seconde, et elles mériteraient d'être formalisées dans un assistant de preuve comme Coq.

La structure de *test interface* peut aussi être utilisée à d'autres fins, par exemple à la définition de bibliothèques, ou de « modules » dans la terminologie de Java 9+.

6 Bibliographie

- [1] Clément AUBERT et Daniele VARACCA. « Processes Against Tests : On Defining Contextual Equivalences ». In : *Journal of Logical and Algebraic Methods in Programming* (2022), p. 100799. ISSN : 2352-2208. DOI : <https://doi.org/10.1016/j.jlamp.2022.100799>. URL : <https://www.sciencedirect.com/science/article/pii/S2352220822000529>.
- [2] Atsushi IGARASHI, Benjamin C. PIERCE et Philip WADLER. « Featherweight Java : A Minimal Core Calculus for Java and GJ ». In : *ACM Trans. Program. Lang. Syst.* 23.3 (mai 2001), p. 396-450. ISSN : 0164-0925. DOI : [10.1145/503502.503505](https://doi.org/10.1145/503502.503505).
- [3] Edlira KUCI et al. « A Co-contextual Type Checker for Featherweight Java ». In : *31st European Conference on Object-Oriented Programming (ECOOP 2017)*. Sous la dir. de Peter MÜLLER. T. 74. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany : Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 18 :1-18 :26. ISBN : 978-3-95977-035-4. DOI : [10.4230/LIPIcs.ECOOP.2017.18](https://doi.org/10.4230/LIPIcs.ECOOP.2017.18). URL : <http://drops.dagstuhl.de/opus/volltexte/2017/7262>.
- [4] Luigi LIQUORI et Arnaud SPIWACK. « Extending FeatherTrait Java with Interfaces ». In : *Theoretical Computer Science*. Theoretical Computer Science 30.1-3 (mai 2008), p. 243-260. DOI : [10.1016/j.tcs.2008.01.051](https://doi.org/10.1016/j.tcs.2008.01.051). URL : <https://hal.inria.fr/inria-00432540>.
- [5] Davide SANGIORGI et David WALKER. *PI-Calculus : A Theory of Mobile Processes*. USA : Cambridge University Press, 2001. ISBN : 0521781779.

Appendices

A Classes et Notations en FeatherweightJava

A.1 Les booléens

```
class Bool extends Object {
  Bool() {
    super();
  }
  Object ite(Object oTrue, Object oFalse){
    return oTrue;
  }
}
class OBool extends Bool {
  OBool() {
    super();
  }
  Object ite(Object oTrue, Object oFalse){
    return oFalse;
  }
}
```

De plus, on note les constantes suivantes :

true = new Bool()

false = new OBool()

A.2 Les entiers

```
class Int extends Object {
  Int() {
    super();
  }
  Bool isZero(){
    return true;
  }
  Bool gt(Int other){
    return false;
  }
  Bool lt(Int other){
    return other.isZero().ite(false,true);
  }
}
class SInt extends Int {
  Int prec;
  SInt(Int prec) {
    super();
    this.prec = prec;
  }
  Bool isZero(){
    return false;
  }
  Bool gt(Int other){
    return other.isZero().ite(true,this.prec.gt(((SInt)other).prec));
  }
  Bool lt(Int other){
    return other.isZero().ite(false,this.prec.lt(((SInt)other).prec));
  }
}
```

On ajoute aussi la notation des nombres :

$0 = \text{new Int}()$

$n = \text{new SInt}(n-1)$

A.3 Les listes

```
// Listes Infinies
class IList extends Object {
  IList next;
  Object obj;
  IList(IList next, Object obj) {
    super();
    this.next = next;
    this.obj = obj;
  }
}

// Listes Finies
class List extends Object {
  List() {
    super();
  }
  Bool isEmpty(){
    return true;
  }
  Object ensureFinite(Object obj){
    return obj;
  }
}

class Cons extends List {
  List queue;
  Object head;
  Cons(List queue, Object head) {
    super();
    this.queue = queue;
    this.head = head;
  }
  Bool isEmpty(){
    return false;
  }
  Object ensureFinite(Object obj){
    return this.queue.ensureFinite(obj);
  }
}

// Plus une méthode ajoutée dans la classe Static
List ensure(List in) {
  return in.isEmpty().ite((List)in, (Cons)in);
}
```

On ajoute aussi les notations des listes :

$[] = \text{new List}()$

$o :: e = \text{new Cons}(e, o)$

Cette dernière méthode assure s'assure que la liste donnée en paramètre est une « vrai » liste, c'est à dire, une liste créée avec [] et ::. La fonction ne se réduit pas en une valeur le cas contraire.

A.4 Les Optional

```
class Optional extends Object {
  Optional() {
    super();
  }
  Object orElse(Object e){
    return e;
  }
}
class Some extends Optional {
  Object value;
  Some(Object value) {
    super();
    this.value = value;
  }
  Object orElse(Object e){
    return this.value;
  }
}
class None extends Optional {
  None() {
    super();
  }
}
```

Ces trois classes définissent la notion d'Optional qui fonctionne comme a' option en OCaml, et qui peut contenir une valeur ou non. La méthode orElse permet de récupérer la valeur contenue si elle est présente, ou de renvoyer le paramètre dans le cas contraire.

B Exemple sur la nécessité d'utiliser une *class table* de test

Nous présentons l'exemple suivant :

```

class IListProblem extends Object {
  IList getList() {
    return new IList(new IList(getList(), 1), 0);
  }
  // Premiere class table
  Bool contains(IList arg) {
    return ((Int)arg.head).eq(0).ite(true, this.contains(arg.tail));
  }
  // Seconde class table
  Bool contains(IList arg) {
    return ((Int)arg.head).eq(1).ite(true, this.contains(arg.tail));
  }
}

```

Nous y ajoutons bien sûr la définition de `IList`, de `Int` et de `Bool` présentées [Annexe A](#).

Dans cet exemple, nous avons deux *class table* qui définissent toutes deux une méthode `getList` qui renvoie intuitivement une liste infinie alternant 0 et 1.

La différence entre ces deux classes réside dans la méthode `IListProblem.contains` qui dans une *class table* vérifie si la liste infinie contient un 0, et dans l'autre vérifie si la liste infinie contient un 1.

Sans ajouter de classes de « test », il n'est pas possible de différencier ces deux *class tables*. En effet, le seul moyen de les différencier serait d'obtenir un résultat différent par la fonction `IListProblem.contains` en fournissant le même paramètre de type `IList`. Cependant, la classe `IList` ayant un paramètre de son propre type, il n'est possible de créer une expression typée par `IList` que en utilisant un appel à méthode récursif. Or, le seul appel qu'il est possible de faire dans ces deux *class tables* est sur `IListProblem.getList`. Et donc, toute liste infinie que l'on pourra créer dans ces deux *class tables* sera nécessairement terminée par la liste infinie de 0 et de 1. Donc les méthodes `contains` renverrons toujours `true`.

Notez que le problème est le même sans la présence de la méthode `getList`, on n'aurait alors pas pu discriminer les deux méthodes `contains` pour un même argument, puisqu'il n'aurait alors pas été possible d'instancier un argument.

Bien sûr, avec une *class table* de tests, il est possible de créer des objets `IList` de la forme que l'on souhaite, et donc de discriminer ces deux méthodes, par exemple avec une liste infinie de 0.

C Preuve du théorème de cohérence du typage des *test interfaces*

Cette section a pour but de présenter la preuve complète du [Théorème 1](#), sur la cohérence du typage des *test interfaces*.

Nous avons \mathcal{T} une *test interface*, \mathcal{C} une *class table* telle que $\mathcal{A} \triangleright \mathcal{T}$ et (\mathcal{B}, e) un programme test, tel que $\mathcal{B} \text{ OK IN } \mathcal{T}$.

Nous allons donc essayer de montrer, par induction sur la preuve de la prémisse, la propriété suivante

$$\mathfrak{T}, \mathcal{B}, \Gamma \Vdash e : D \implies \exists C \quad \mathcal{A} \oplus \mathcal{B}, \Gamma \vdash e : C \quad \text{et} \quad C <: D \quad (1)$$

C.1 Propriétés sur les fonctions utilitaires

La définition de \vdash dans Featherweight Java de base utilise les fonctions utilitaires non surchargées. Afin de bien les différencier, nous allons noter avec un prime les versions surchargées définies [FIGURE 8](#).

Par exemple, (fields') correspond à la définition surchargée dans la *test interface* \mathfrak{T} et la *class table* de tests \mathcal{B} , et (fields) correspond à la définition originale dans la *class table* $\mathcal{A} \oplus \mathcal{B}$. Il en va de même pour l'opérateur $(<:')$, qui dénotera la clôture transitive et réflexive de l'union des relations $<:$ sur \mathfrak{T} et extends dans \mathcal{B} et l'opérateur $(<:)$ qui dénotera la relation de sous-typage sur la *class table* $\mathcal{A} \oplus \mathcal{B}$.

Nous allons déjà montrer les trois propriétés suivantes sur ces fonctions utilitaires :

$$C \text{ f} \in \text{fields}'_{\mathfrak{T}, \mathcal{B}}(Cc) \implies \exists C' <: C \quad C' \text{ f} \in \text{fields}_{\mathcal{A} \oplus \mathcal{B}}(Cc) \quad (2)$$

$$\text{mtype}'_{\mathfrak{T}, \mathcal{B}}(m, Cc) = \bar{D} \rightarrow C \implies \exists C' <: C \quad \exists \bar{D}' :> \bar{D} \quad \text{mtype}_{\mathcal{A} \oplus \mathcal{B}}(m, Cc) = \bar{D}' \rightarrow C' \quad (3)$$

$$\text{construct}'_{\mathfrak{T}, \mathcal{B}}(Cc) = \bar{D} \implies \exists \bar{D}' :> \bar{D} \quad \exists \bar{f} \quad \text{fields}_{\mathcal{A} \oplus \mathcal{B}}(Cc) = \bar{D}' \bar{f} \quad (4)$$

Propriété 2 Montrons cette propriété par induction sur la relation $<:'$.

Nous pouvons déjà discriminer sur le cas suivant :

Cas $C \text{ f} \in \bigcup_{C <: E} \text{fields}'_{\mathfrak{T}, \mathcal{B}}(E)$ On applique alors directement l'hypothèse d'induction.

Sinon En inversant la règle qui permet de prouver $C \text{ f} \in \text{fields}'_{\mathfrak{T}, \mathcal{B}}(Cc)$, il y reste trois possibilités, correspondant aux trois règles de construction de fields . La première discrimination permet d'éliminer les second cas où l'attribut aurait été défini pour une classe mère.

Cas $[Cc \{C \text{ f}, \dots\}] \in \mathfrak{T}$ (**LTi-ATTR**)

Alors, en inversant la preuve de $\mathcal{A} \triangleright \mathfrak{T}$, on obtient une seule règle possible : la règle (**TRI-ATTR**), et on a donc

$$C' \text{ f} \in \text{fields}_{\mathcal{A}}(Cc) \quad \text{et} \quad C' <: C$$

Cas $[Cc (\dots, C \text{ f}, \dots)] \in \mathfrak{T}$ (**LTi-ATTRC**)

Alors, en inversant la preuve de $\mathcal{A} \triangleright \mathfrak{T}$, on obtient une seule règle possible : la règle (**TRI-CSTR**), et on a donc, puisque le field f est défini,

$$C' \text{ f} \in \text{fields}_{\mathcal{A}}(Cc) \quad \text{et} \quad C' <: C$$

Cas $\text{class } Cc \{C \text{ f}; \dots\} \in \mathcal{B}$ (**LCT-ATTR**)

Alors, par construction de fields , on a $C \text{ f} \in \text{fields}_{\mathcal{A} \oplus \mathcal{B}}(Cc)$ (On prend $C' = C$)

Propriété 3 Procédons par induction sur la définition de $\text{mtype}'_{\mathfrak{T}, \mathcal{B}}$.

Nous avons comme hypothèse $\text{mtype}'_{\mathfrak{T}, \mathcal{B}}(m, Cc) = \bar{D} \rightarrow C$.

Cas $[Cc : C \text{ m}(\bar{D})] \in \mathfrak{T}$ (**LTi-METHC**)

Alors, puisque $\mathcal{A} \triangleright \mathfrak{T}$, c'est nécessairement par la règle (**TRI-METH**), et on a donc.

$$\text{mtype}_{\mathcal{A}}(\text{m}, C) = \bar{D}' \rightarrow C' \quad ; \quad \bar{D} <: \bar{D}' \quad ; \quad C' <: C$$

Cas $\text{class } Cc \{ \dots C \text{ m}(\bar{D} \bar{x}) \} \in \mathcal{B}$ (**LCT-METH**)

Alors, par définition de mtype , on a que

$$\text{mtype}_{\mathcal{B}}(\text{m}, Cc) = \bar{D} \rightarrow E \quad \text{et} \quad Cc <: Cc$$

Cas $\text{mtype}(\text{m}, C) = \bar{D} \rightarrow E$ **et** $C <: Cc$ (**LUP-METH**)

Puisque $C <: Cc$, alors, il existe une chaîne de classes

$$C = C_0, C_1, \dots, C_n = Cc$$

telles que C_k extends C_{k+1} dans \mathcal{A} ou \mathcal{B} .

Or, on sait que les définitions de \mathcal{A} et \mathcal{B} sont OK (c'est imposé par la définition de $\mathcal{A} \triangleright \mathfrak{T}$). Être OK implique notamment que l'on ne change pas le type d'une méthode que l'on aurait éventuellement surchargée. On a donc que pour chacune de ces classes :

$$\text{mtype}_{\mathcal{A} \oplus \mathcal{B}}(\text{m}, C_n) = \text{mtype}_{\mathcal{A} \oplus \mathcal{B}}(\text{m}, C_{n-1}) = \dots = \text{mtype}_{\mathcal{A} \oplus \mathcal{B}}(\text{m}, C_0)$$

Donc $\text{mtype}(\text{m}, Cc) = \bar{D} \rightarrow E$.

Propriété 4 Inversons la règle qui a permis de prouver :

$$\text{construct}'_{\mathfrak{T}, \mathcal{B}}(Cc) = \bar{D}$$

Cas $[Cc (\bar{D} \bar{f})] \in \mathfrak{T}$ (**LTi-CSTR**)

Alors, puisque $\mathcal{A} \triangleright \mathfrak{T}$, c'est nécessairement par la règle (**TRI-CSTR**), et on a donc

$$\exists \bar{D}' \quad \bar{D}' \bar{f} \bar{O} = \text{fields}_{\mathcal{A}}(C)$$

Cas $\text{class } Cc \{ Cc(\bar{D} \bar{f}) \{ \dots \} \dots \} \in \mathcal{B}$ (**LCT-CSTR**)

Alors, puisqu'en Featherweight Java, les objets sont immuables et les constructeurs définissent tous les fields, on sait que les paramètres du constructeurs sont *exactement* les champs de la classe. Et donc

$$\text{fields}_{\mathcal{A} \oplus \mathcal{B}}(Cc) = \bar{D} \bar{f} \quad \text{et} \quad \bar{D} <: \bar{D}$$

C.2 Corps de la preuve

Démarrons l'induction sur la preuve de la prémisse de la **Propriété 1**, présentée au début de la section. Cette prémisse est la suivante :

$$\mathfrak{T}, \mathcal{B}, \Gamma \Vdash e : D$$

Je vais, pour chaque règle, lister directement en dessous la liste des hypothèses qu'elle induit.

(TI-VAR)

1. $e = x$
2. $C = \Gamma(x)$
3. $x \in \text{dom}(\Gamma)$

Alors, d'après la règle FJ T-VAR

$$\mathcal{A} \oplus \mathcal{B}, \Gamma \vdash x : \Gamma(x)$$

(TI-FIELD)

1. $e = e_0 . f$
2. $C = C$
3. $\mathfrak{T}, \mathcal{B}, \Gamma \Vdash e_0 : Cc$ donc $\mathcal{A} \oplus \mathcal{B} \vdash e_0 : Cc'$ et $Cc' <: Cc$
4. $C f \in \text{fields}'_{\mathfrak{T}, \mathcal{B}}(Cc)$

On peut alors appliquer la [Propriété 2](#) sur la quatrième hypothèse qui nous donne

$$\exists C' <: C \quad C' f \in \text{fields}(Cc)$$

Alors, on peut directement appliquer la règle FJ T-FIELD

(TI-INVK)

1. $e = e_0 . m(\overline{e_x})$
2. $C = C$
3. $\mathfrak{T}, \mathcal{B}, \Gamma \Vdash e_0 : Cc$ donc $\mathcal{A} \oplus \mathcal{B} \vdash e_0 : Cc'$ et $Cc' <: Cc$
4. $\mathfrak{T}, \mathcal{B}, \Gamma \Vdash \overline{e_x} : \overline{CX}$ donc $\mathcal{A} \oplus \mathcal{B} \vdash \overline{e_x} : \overline{CX'}$ et $\overline{CX'} <: \overline{CX}$
5. $\text{mtype}'_{\mathfrak{T}, \mathcal{B}}(m, Cc) = \overline{D} \rightarrow C$
6. $\overline{CX} <: \overline{C}$

On peut alors appliquer la [Propriété 3](#) sur la cinquième hypothèse, on obtient que

$$\text{mtype}_{\mathcal{A} \oplus \mathcal{B}}(m, Cc) = \overline{D'} \rightarrow C' \quad \text{et} \quad \overline{D} <: \overline{D'} \quad \text{et} \quad C' <: C$$

Alors, en utilisant la même preuve que pendant la preuve de la [Propriété 3](#) (après inversion de la règle (LUP-METH))

$$\text{mtype}_{\mathcal{A} \oplus \mathcal{B}}(m, Cc') = \overline{D'} \rightarrow C' \quad \text{et} \quad \overline{CX'} <: \overline{CX} <: \overline{D} <: \overline{D'}$$

On peut maintenant appliquer la règle FJ T-INVK et obtenir le résultat.

(TI-NEW)

1. $e = \mathbf{new} \ C(\overline{e_x})$
2. $C = \overline{C}$
3. $\mathfrak{T}, \mathcal{B}, \Gamma \Vdash \overline{e_x} : \overline{CX}$ donc $\mathcal{A} \oplus \mathcal{B} \vdash \overline{e_x} : \overline{CX'}$ et $\overline{CX'} <: \overline{CX}$
4. $\text{construct}'_{\mathfrak{T}, \mathcal{B}}(C) = \overline{D}$
5. $\overline{CX} <: \overline{D}$

On peut appliquer la [Propriété 4](#) sur la quatrième hypothèse. On obtient alors que :

$$\text{fields}_{\mathcal{A} \oplus \mathcal{B}}(C) = \overline{D'} \ \overline{f} \quad \text{et} \quad \overline{CX'} <: \overline{CX} <: \overline{D} <: \overline{D'}$$

On peut maintenant appliquer la règle FJ T-NEW et obtenir le résultat.

(TI-UCAST)

1. $e = (C)e_0$
2. $C = \overline{C}$
3. $\mathfrak{T}, \mathcal{B}, \Gamma \Vdash e_0 : C$ donc $\mathcal{A} \oplus \mathcal{B} \vdash e_0 : D$ et $D <: C$
4. $D <: C$

On a alors les relations de sous-typage suivante :

$$D' <: D <: C$$

On peut donc appliquer directement la règle T-UCAST

(TI-DCAST)

Exactement la même preuve que le point précédent en appliquant la règle T-DCAST

D Preuve de la [Propriété 1](#) sur les valeurs infinies

On rappelle tout d'abord la propriété :

$$\forall h \forall \alpha \quad h[\alpha] \rightarrow^* e' \implies \exists \beta \quad e' = h[\beta]$$

Nous allons d'abord montrer la propriété sur les expressions suivante :

Propriété 4 Soit e et f des expressions fermées, telles que $e \rightarrow^* f$,

Si $e = \mathbf{new} \ C(\overline{e})$

Alors $f = \mathbf{new} \ C(\overline{f})$ et $\overline{e'} \rightarrow^* \overline{f'}$.

Montrons cette propriété par récurrence sur la longueur de la réduction $e \rightarrow^* f$.

Initialisation

Dans le cas $e = f$, on a simplement $f = \mathbf{new} \ C(\overline{e})$ et $\overline{e'} \rightarrow^0 \overline{e'}$.

Hérédité

Nous sommes dans l'hypothèse où $e \rightarrow^n f$. Observons la première relation. Elle se fait nécessairement sur un nœud de type [E-METH](#), [E-FIELD](#) ou [E-CAST](#), qui est donc contenu dans l'un des \bar{e}' . Donc, on a $e'_i \rightarrow e''_i$. On note $e''_k = e'_k$ pour k différent de i , et on obtient que

$$e = \mathbf{new} \ C(\bar{e}) \rightarrow \mathbf{new} \ C(\bar{e}) \rightarrow^{n-1} f$$

On peut donc appliquer l'hypothèse de récurrence à la dernière relation \rightarrow^{n-1} , et nous obtenons bien que

$$f = \mathbf{new} \ C(\bar{f}) \quad \text{et} \quad \bar{e}' \rightarrow^* \bar{e}'' \rightarrow^* \bar{f}'$$

La récurrence est terminée.

Nous allons donc pouvoir facilement montrer la [Propriété 1](#) par induction sur le paramètre \bar{h} .

$\bar{h} = x$

En notant β tel que $\beta(x) = e'$, on obtient que

$$\bar{h}[\alpha] = \alpha(x) \rightarrow e' = \bar{h}[\beta]$$

$\bar{h} = \mathbf{new} \ C(\bar{h})$

Alors, par application de la [Propriété 4](#), on obtient que

$$e' = \mathbf{new} \ C(\bar{e}) \quad \text{et} \quad \overline{\bar{h}[\alpha]} \rightarrow^* \bar{e}$$

On peut alors appliquer l'hypothèse d'induction, et obtenir que

$$\bar{e} = \overline{\bar{h}[\beta_x]}$$

Donc, en notant β la concaténation des β_x , qui est possible puisque les noms de variables utilisés dans \bar{h} sont tous distincts, on obtient le résultat

$$e' = \bar{h}[\beta]$$

E Preuve du *context lemma* avec valeurs infinies

Nous allons démontrer une version un peu plus puissante de ce théorème, ce qui simplifiera l'induction que nous ferons.

Tout d'abord, nous allons définir quelques notations locales à cette preuve.

Déjà, nous ne spécifierons pas la *class table*, puisque nous supposerons que nous travaillerons dans la *class table* $\mathcal{A} \oplus \mathcal{B}$, car puisque le \oplus impose que les noms de classe n'entrent pas en collision, si un expression s'exécutant dans \mathcal{A} ou \mathcal{B} s'exécutera de la même façon dans $\mathcal{A} \oplus \mathcal{B}$.

La lettre h dénotera une expression FJ quelconque (ouverte ou fermée). Nous les dénoterons ainsi afin de ne pas les confondre avec les expressions « fermées », dénotées elles e, f .

Nous noterons aussi ε et γ les *mappings* servant à « remplir » les expression ouvertes, afin de ne pas les confondre avec α et β , qui servent à remplir les valeurs avec variables.

On note toujours le préordre entre deux expressions fermées de la même façon :

$$e \prec f \quad \text{ssi} \quad \forall h \quad (\exists \alpha \quad e \rightarrow \bar{h}[\alpha]) \implies (\exists \beta \quad f \rightarrow \bar{h}[\beta])$$

Nous allons aussi étendre cette définition à deux *mappings* ayant le même domaine :

$$\varepsilon \prec \gamma \quad \text{ssi} \quad \forall x \in \text{Dom}(\varepsilon) = \text{Dom}(\gamma) \quad \varepsilon(x) \prec \gamma(x)$$

Le théorème, plus général que nous allons montrer est le suivant :

Théorème 2 *Pour ε, γ des mappings de même domaine*

Pour h expression avec des variables libres tel que $\text{Var}(h) \subset \text{Dom}(\varepsilon)$

Si $\varepsilon \prec \gamma$

Alors $h[\varepsilon] \prec h[\gamma]$

C'est à dire

$$\forall h (\exists \alpha \quad h[\varepsilon] \rightarrow^* h[\alpha]) \implies (\exists \beta \quad h[\gamma] \rightarrow^* h[\beta])$$

C'est sous cette seconde forme que nous allons prouver le théorème (elle est plus « bas niveau »).

Nous allons tout d'abord faire une récurrence sur la longueur de la réduction $h[\varepsilon] \rightarrow^* h[\beta]$, notée n .

Initialisation On démontre alors le résultat par induction sur h .

On suppose que $h[\varepsilon] = h[\alpha]$.

Cas $h = [\cdot]$

Alors

$$h[\gamma] = h[\beta] \quad \text{avec} \quad \beta = (h[\gamma])$$

Cas $h = \text{new } C(\bar{h})$

Alors, il y a deux possibilités.

Ou bien, on a $h = x$, et alors puisque $\varepsilon \prec \gamma$ et que $\varepsilon(x) = h[\alpha]$, alors, on a que $\gamma(x) \rightarrow^* h[\beta]$

Ou bien, on a $h = \text{new } C(\bar{h})$, et donc par induction

$$\bar{h}'[\gamma] \rightarrow^* \bar{h}'[\beta']$$

Et donc, en notant β la concaténation des $\bar{\beta}'$ (puisque les noms de variables sont utilisés de manière unique, les domaines de définition sont donc disjoints), on obtient que

$$h[\gamma] \rightarrow^* h[\beta]$$

Hérédité Notre hypothèse de récurrence (HR) est la suivante :

$$\forall h \quad \varepsilon \prec \gamma \implies h[\varepsilon] \prec^{<n} h[\gamma]$$

Nous aurons ensuite trois cas, suivant l'endroit où s'effectue la première étape de la réduction.

La réduction se fait uniquement avec des nœuds de h Alors, on a $h \rightarrow h'$ indépendamment du *mapping* considéré, et on peut appliquer (HR) qui nous donne que, puisque $h'[\varepsilon] \rightarrow^{n-1} h[\alpha]$

$$h[\gamma] \rightarrow h'[\gamma] \rightarrow^* h[\beta]$$

La réduction se fait sur un nœud à l'intérieur d'une variable x de h Alors, dans cette occurrence de la variable x dans h , nous avons une réduction de la forme suivante.

$$\varepsilon(x) \rightarrow e^x$$

Choisissons alors un nom de variable non utilisé dans ϵ et $\gamma : y$, et créons ϵ' ainsi :

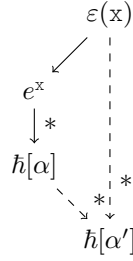
$$\epsilon' = \begin{bmatrix} x & \mapsto & \epsilon(x) \\ y & \mapsto & e^x \\ v & \mapsto & \epsilon(v) \end{bmatrix} \quad \gamma' = \begin{bmatrix} x & \mapsto & \gamma(x) \\ y & \mapsto & \gamma(x) \\ v & \mapsto & \gamma(v) \end{bmatrix}$$

Ainsi, la réduction considérée s'exprime comme

$$h[\epsilon] \rightarrow h'[\epsilon'] \rightarrow^{n-1} h[\alpha]$$

Or, nous avons aussi que $\epsilon' \prec \gamma'$. Nous n'avons en effet que $\epsilon(y) \prec \gamma(y)$ à vérifier, c'est à dire que $\epsilon(x) \rightarrow e^x \prec \gamma(x)$

Cela est vrai par propriété de confluence sur l'opération de réduction. Intuitivement, un nœud d'une expression FJ ne peut se réduire que d'une seule façon (l'une des trois règles de réduction), et cette réduction se fait sans perte d'information (dans le sens où on aurait pas pu en extraire plus sans modifier le nœud en question). Il suffit donc de réduire les mêmes nœuds des deux côtés afin d'obtenir une même expression.



On peut alors appliquer l'hypothèse $\epsilon \prec \gamma$.

Nous avons donc toutes les conditions réunies pour appliquer l'hypothèse de récurrence.

La réduction s'est faite entre un nœud de h et un nœud de ϵ Il y a trois cas de figure pour lesquels les deux cas précédents ne sont pas adéquats. Le premier est un appel à *field* de la forme $x.field$, le second est un appel à méthode de la forme $x.meth(\bar{h})$ et le troisième cas est le *cast* de la forme $(D)x$.

Dans ces trois cas, nous avons nécessairement $\epsilon(x) = \mathbf{new} \ C(\overline{e^\epsilon})$, et donc aussi $\gamma(x) = \mathbf{new} \ C(\overline{e^\gamma})$ avec $\overline{e^\epsilon} \prec \overline{e^\gamma}$, puisque $\epsilon \prec \gamma$.

Dans le cas de $x.field$, on choisit un nouveau nom de variable inutilisé y , et nous créons une nouvelle fois ϵ' et γ' ainsi.

$$\epsilon' = \begin{bmatrix} x & \mapsto & \epsilon(x) \\ y & \mapsto & e_k^\epsilon \\ v & \mapsto & \epsilon(v) \end{bmatrix} \quad \gamma' = \begin{bmatrix} x & \mapsto & \gamma(x) \\ y & \mapsto & e_k^\gamma \\ v & \mapsto & \gamma(v) \end{bmatrix}$$

Nous avons alors toujours $\epsilon' \prec \gamma'$, mais aussi que la première réduction s'écrit

$$h[\epsilon] \rightarrow h'[\epsilon'] \rightarrow^{n-1} h[\alpha]$$

où h' est obtenu en remplaçant l'occurrence de $x.field$ susnommée par y .

On peut donc appliquer l'hypothèse de récurrence pour obtenir, puisque les *class table* considérées sont les mêmes

$$h[\gamma] \rightarrow h'[\gamma'] \rightarrow^* h[\beta]$$

Dans le cas de $x.meth(\bar{h})$, la procédure est presque la même.

Nous allons noter \bar{v} les noms des paramètres de $C.meth$. Nous noterons aussi h_m le corps de cette méthode.

Nous allons enfin obtenir h'_m en remplaçant dans h_m la variable **this** par la variable x et les appels à chaque variable v_k par l'expression à variables h'_k .

Nous pouvons garder ε et γ tels qu'ils sont.

Alors, en remplaçant l'appel $x.meth(\bar{h})$ surnommé par h'_m dans h , on obtient une nouvelle expression h' telle que la réduction soit

$$h[\varepsilon] \rightarrow h'[\varepsilon] \rightarrow^{n-1} h[\alpha]$$

Mais en ayant aussi que

$$h[\gamma] \rightarrow h'[\gamma]$$

On peut enfin appliquer l'hypothèse de récurrence de la même manière.

Enfin, dans le dernier cas (D)x, puisque la réduction s'est bien effectuée, c'est que $C <: D$. On peut donc simplement remplacer dans h le cast par la variable x seule.

F Le stage au LACL

Je vais présenter ici le laboratoire dans lequel j'ai fait ce stage. Le Laboratoire d'Algorithmique, Complexité et Logique est un laboratoire inclus dans l'Université Paris-Est Créteil. Le laboratoire est donc dans un campus universitaire, et accueille aussi quelques doctorant · es, que je n'ai que peu vu · es à cause de la période tardive du stage. Ce stage m'a en partie confirmé que je voulais m'orienter vers de la recherche, le travail bien que modeste que j'ai effectué correspondait à l'image que je me faisais de l'« exercice » de la recherche, ce que je n'avais jamais expérimenté avant. Je reste conscient que je n'ai vu qu'une très petite partie du quotidien d'un · e chercheur · euse. J'ai aussi été sous la tutelle de Clément Aubert, Maître de conférence à l'université d'Augusta, Géorgie. Je n'ai pas beaucoup croisé les membres des autres équipes du laboratoire, ne serait-ce que pendant quelques discussions lors des pauses repas.