



Examen – Master Recherche Informatique

Module MMPFPSD
Mardi 18 janvier 2005

Correction

Consignes

Tout document est autorisé !

Cet examen est prévu pour durer deux heures (2h). Son but est de vérifier votre compréhension, veuillez donc à ce que vos explications soient claires et concises.

Merci d'avance et bon travail !

Exercice 1 (*La liaison dynamique en Java*) [10 mn – 1,7/20]

Lors du passage de la version 1.3 à la version 1.4, la sémantique du langage Java a légèrement évolué pour adopter celle de Visual Basic (sans aucune publicité semble-t-il!). La signature (au sens introduit dans le cours) est passée du tableau 1(a) au tableau 1(b). La case de la 6^e ligne, 2^e colonne a changé.

▷ **Commentez et expliquez.**

Cette case correspond au cas d'un récepteur Down, déclaré Down, et d'un attribut Bottom. 2 méthodes seraient susceptibles de rendre le service :

- 1. `ctv(Bottom)` définie dans Top - récepteur moins spécialisé (`Down<:Top`), mais argument bien spécialisé (`Bottom`)**
- 2. `ctv(Middle)` définie dans Down - récepteur bien spécialisé (`Down`), mais argument moins spécialisé (`Bottom<:Middle`).**

Les versions de Java antérieures à la 1.3 refusaient de faire un choix. Un message d'erreur indiquait une « ambiguïté. » À partir de la version 1.4, le choix s'est porté sur la première solution, donnant priorité à la spécialisation de l'argument sur la spécialisation du récepteur. C'est un choix qui n'a pas eu de conséquence sur le code existant puisque précédemment cette situation était rejetée !

appels	u	d	ud
cv(t)	Up	Up	Up
cv(m)	Up	Down	Up
cv(b)	Up	Down	Up
ctv(t)	Erreur	Erreur	Erreur
ctv(m)	Erreur	Down	Erreur
ctv(b)	Up	<i>Erreur</i>	Up

(a) Résultats jusqu'à Java 1.3

appels	u	d	ud
cv(t)	Up	Up	Up
cv(m)	Up	Down	Up
cv(b)	Up	Down	Up
ctv(t)	Erreur	Erreur	Erreur
ctv(m)	Erreur	Down	Erreur
ctv(b)	Up	<i>Up</i>	Up

(b) Résultats depuis Java 1.4

TAB. 1 – Les résultats des liaisons dynamique de Java

Exercice 2 (*Médium de vote*)

[20 mn – 3,3/20]

On considère un composant de communication (médium) de vote. Sa spécification informelle est la suivante :

« Le médium de vote permet à un composants de proposer un vote à la fois. Un vote est une liste de choix qui est proposé pendant une durée définie par le composant qui propose le vote. Les composants qui participent aux votes peuvent faire un choix dans la liste. Une fois ce choix fait, il peut être remis en cause tant que le vote n'est pas clos. Une fois le vote clos, les votes sont ignorés jusqu'au prochain vote. »

Question 1

- ▷ Décrivez par un diagramme UML de collaboration de niveau spécification la structure du médium de vote.

Il fallait dessiner un diagramme de collaboration faisant apparaître :

- les rôles *Votant* et "*Proposeur*" de vote,
- la classe *Médium de vote*,
- les *relations* entre la classe médium et les rôles,
- les *interfaces* offertes par le médium et celles requises (une méthode suffisait à chaque fois),
- les relations (*implements* ou *uses*) entre les interfaces et les rôles.

Vous trouverez une description détaillée d'un médium de vote à l'adresse suivante :

<http://www-info.enst-bretagne.fr/medium/specification/vote.html>

Il était inutile d'aller à ce niveau de détail, mais suggérer chacun des éléments était important.

Question 2

- ▷ Donnez un exemple de contrat pour chacun des 4 niveaux définis dans le cours (syntaxique, sémantique, synchronisation et qualité de service)¹.
- *Syntaxique* : Il suffisait de donner une signature de méthode par exemple :
`vote(int choix)`
 - *Sémantique* : On pouvait préciser que la précondition de la méthode `vote` est que l'on a pas encore voté (un booléen à faux), et sa post-condition, qu'on a voté (ce booléen est passé à vrai).
 - *Synchronisation* : On pouvait dire qu'on ne pouvait pas appeler la méthode `vote(int)` tant que le vote n'avait pas démarré (méthode `lancerVote` du rôle "Proposeur" de vote par exemple) et après sa fin (si il y avait un délai de vote).
 - *Qualité de service* : On pouvait dire par exemple que tout vote réalisé était pris en compte (fiabilité) – ce qui n'est pas forcément facile à réaliser ! – lorsque le résultat final est transmis.

Exercice 3 (La généricité en Java 1.5)

[30 mn – 5/20]

La généricité a été ajoutée en Java 1.5. Le but de ce problème est de l'étudier. Vous n'avez pas besoin d'être un expert Java pour répondre aux questions, si vous ne pouvez pas produire d'exemple en Java, faites le dans un pseudo langage compréhensible.

La forme de polymorphisme paramétrique ajouté en Java est la paramétrisation des classes et des méthodes par des types. Par exemple, la classe `C` peut être paramétrée par la classe `G` en utilisant la syntaxe suivante `C<G>`.

Le principal bénéfice visible de la généricité réside dans une utilisation plus simple des collections :

```
List<String> maListe = new List<String>();
maListe.add("toto");
String resultat = maListe.get(0);
```

au lieu de :

```
List maListe = new List();
maListe.add("toto");
String resultat = (String) maListe.get(0);
```

Question 1

- ▷ La simplicité est-elle le seul intérêt ? Justifier (et illustrer par un exemple) votre réponse.

Non, le principal intérêt de la généricité est la sûreté. En effet, le compilateur peut dorénavant vérifier le type (réel) des données mises dans la liste. Par exemple, dans le cas d'une liste générique le code `maListe.add(new Object());` est rejeté par le compilateur.

¹On acceptera tout "langage" de description de contrat.

Question 2

Soit l'interface générique Java suivante :

```
public interface List<E> {
    void add(E x);
    E get(int index);
    int size();
}
```

Supposons, de plus, que la classe `LinkedList<E>` réalise l'interface `List<E>`.

Dans le code suivant :

```
List<String> ls = new LinkedList<String>();
List<Object> lo = ls;
```

la seconde ligne de code ne peut pas être légale, sinon elle contredirait la correction du système de type de Java.

- ▷ Produisez un code qui prouve cette affirmation et expliquer le.

Si la seconde ligne était valide, alors il serait possible de créer un code (voir ci-dessous) qui ajoute un objet de type `Object` dans `lo` et donc dans `ls`. Cet objet pourrait ensuite être utiliser comme une chaîne de caractères (`String`) ce qu'il n'est pas !

```
lo.add(new Object());
ls.get(0).length();
```

- ▷ Exprimer mathématiquement ce que signifie cette affirmation.

$$\forall C, X, Y <: \text{Object} \quad C<X> <: C<Y> \Rightarrow X = Y$$

Question 3

Dans un tel système, il n'est pas possible de définir une méthode capable de prendre en paramètre toutes les listes. La solution introduite par Sun est l'introduction d'un nouveau type ? appelé type inconnu (*wildcard*) et définit par :

$$\forall X, C <: \text{Object} \quad C<X> <: C<?>$$

- ▷ Quel est alors selon vous le type des élément d'une liste de type `List<?>` ?

Le type inconnu peut être utilisé avec tout type, donc la seule chose que l'on peut supposer est que les éléments d'une liste de type `List<?>` sont des objets, *i.e.* leur type est `Object`.

- ▷ Est-il possible d'ajouter un élément dans une telle liste ? Justifier votre réponse.

Non, du fait de la question 2.

Question 4

En java 1.5 a été ajouté la notion de *foreach*, toute collection peut être parcourue par un code de la forme :

```
for(TypeElement element : collection)
// utilisation de la variable element
```

Si l'on suppose définit les trois classes suivantes :

```
abstract class Dessin { public abstract void afficher(); }

class Cercle extends Dessin { public void afficher() { ... } }

class Rectangle extends Dessin { public void afficher() { ... } }
```

- ▷ On souhaite définir une méthode générique qui prend en paramètre une liste d'élément que l'on peut afficher (ils ont une méthode `afficher`) et qui affiche chaque élément de la liste.

La solution proposée ci-dessous n'est pas assez générale. Expliquer pourquoi. Quelle solution proposeriez-vous ?

```
void afficher(List<Dessin> dl) {
    for(Dessin d : dl)
        d.afficher();
}
```

La méthode `afficher` n'accepte que des listes de dessin (de type `List<Dessin>`) et non pas des listes sous-types. Par exemple, si `Cercle` est un sous-type de `Dessin`, les listes de cercles (de type `List<Cercle>`) ne sont pas acceptées par la fonction.

La solution adopté par Sun est d'ajouter la généricité contrainte, c'est-à-dire que l'on va pouvoir spécifier une contrainte d'héritage lors de l'instanciation générique. Le mot clé choisi par Sun est `extends`. De plus, on peut mixer le type inconnu et cette contrainte, le code de la méthode devient donc :

```
void afficher(List<? extends Dessin> dl) {
    for(Dessin d : dl)
        d.afficher();
}
```

Exercice 4 (*Méta-modèle de composant*) [20 mn – 3,3/20]

Le but de cet exercice est la construction d'un méta-modèle de composant. Il ne faudra pas se contenter d'un diagramme, mais le justifiez en décrivant l'intention que vous avez en introduisant les « concepts ».

Question 1

- ▷ *Esquissez* un méta-modèle de composant. On pourra partir, pour un composant, de la définition suivante d'un composant par Clemens Szyperski :

A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only.

On pouvait dessiner un diagramme de classe faisant apparaître :

- une classe *composant*, – pour introduire le concept
- une classe *interface*,
- une classe *relation* avec deux sous classes *offert* et *requis* – pour décrire les types de relation qui peuvent exister entre un composant et ses interfaces.

On avait traduit tous les concepts introduit par Szyperski. Il existe évidemment d'autres méta-modèles...

Question 2

- ▷ Compléter votre modèle, pour tenir compte de la suite de la définition :

A software component (...) is subject to composition by third parties.

Il fallait exprimer le fait qu'une interface de composant pouvait être relié à une autre interface de composant. On pouvait introduire la classe *connecteur* par exemple.

Là, les choses se compliquent si on veut exprimer des contraintes qui rendent valide ou non l'assemblage ! Mais on ne demandait pas de rentrer à ce niveau de détails... Il faudrait construire un système de type complet (et complexe) !

Question 3

- ▷ Pensez-vous qu'un assemblage de composants soit un composant ? Argumentez...

Il existe des modèles qui considèrent qu'un composant est composé de composants (Fractal, CCM, .NET,...) et d'autres qui l'interdisent (EJB). Ce qui m'intéressait, étaient les arguments développés... une analyse (ou un début d'analyse) des conséquences de ce choix.

Élégance d'un modèle récursif, mais spécification et règles plus complexes. Simplicité d'un modèle plan, mais besoin d'outils d'abstractions...

Exercice 5 (*Le λ -calcul polymorphe*)

[40 mn – 6,7/20]

Comme vu en cours, le λ -calcul permet de modéliser formellement la notion de fonction. Nous allons dans ce problème étudier la notion de polymorphisme dans le cadre du λ -calcul. Dans le cadre de l'exercice, polymorphisme signifie qu'une fonction pourra prendre en paramètre des données de type différents. Plus précisément, nous allons introduire un nouveau constructeur indiquant nous souhaitons construire une valeur (en général, une fonction) polymorphe. Cette construction est notée (dans la tradition) `let` et on parle donc de `let`-polymorphisme.

La syntaxe de notre calcul que nous noterons ML_0 est donc :

$M ::= x$	Une variable
MM	Une application
$\lambda x.M$	Une fonction
let $x = M$ in M	Une définition polymorphe

La sémantique de ML_0 est donnée par les deux règles de réduction (dites β -réductions) suivantes :

$$\lambda x.M_1M_2 \rightarrow [x \mapsto M_2]M_1 \quad (\beta\text{-APP}) \qquad \mathbf{let} \ x = M_1 \ \mathbf{in} \ M_2 \rightarrow [x \mapsto M_1]M_2 \quad (\beta\text{-LET})$$

avec les définitions auxiliaires suivantes :

$$\text{Substitution : } \begin{cases} [x \mapsto M]x = M \\ [x \mapsto M]x' = x' & \text{si } x \neq x' \\ [x \mapsto M]M_1M_2 = [x \mapsto M]M_1[x \mapsto M]M_2 \\ [x \mapsto M]\lambda x'.M' = \lambda x'. [x \mapsto M]M' & \text{si } x \neq x' \text{ et } x' \notin FV(M) \end{cases}$$

$$\text{Variables libres : } \begin{cases} FV(x) = \{x\} \\ FV(M_1M_2) = FV(M_1) \cup FV(M_2) \\ FV(\lambda x.M) = FV(M) \setminus \{x\} \end{cases}$$

$$\text{Renommage des variables liées : } \lambda x.M = \lambda x'. [x'/x]M \quad \text{si } x' \notin FV(M)$$

Question 1

Pour simplifier le problème, on utilisera , ici, une notion intuitive d'équivalence correspondant à « se réduisent de la même façon ».

- ▷ **À quelle application un `let` est-il équivalent (on pourra étudier les règles de β -réductions) ?**

D'après (β -let), $\mathbf{let} \ x = M \ \mathbf{in} \ N \rightarrow [x \mapsto M]N$ et (β -app), $\lambda x.NM \rightarrow [x \mapsto M]N$ se réduisent « de la même façon », donc :

$$\mathbf{let} \ x = M \ \mathbf{in} \ N \equiv \lambda x.NM$$

Commentaires :

- **Le `let` semble donc inutile d'après la réponse à cette question, l'intérêt de cet ajout apparaît lorsque l'on cherche à ajouter le polymorphisme comme nous le verrons plus loin.**

Question 2

- ▷ Les définitions auxiliaires ne prennent pas en compte le `let`. Compléter la définition des notions de substitution, de variables libres et de renommage des variables liées.

À partir, de l'équivalence de la question précédente, et des définitions pour l'application et la lambda abstraction, on peut déduire que :

– Substitution :

$$[x \mapsto M]\text{let } x' = M_1 \text{ in } M_2 = \text{let } x' = [x \mapsto M]M_1 \text{ in } [x \mapsto M]M_2 \text{ si } x \neq x' \text{ et } x' \notin FV(M)$$

– Variables libres :

$$FV(\text{let } x = M_1 \text{ in } M_2) = FV(M_1) \cup (FV(M_2) \setminus \{x\})$$

– Renommage des variables liées :

$$\text{let } x = M_1 \text{ in } M_2 = \text{let } x' = M_1 \text{ in } [x \mapsto x']M_2 \text{ si } x' \notin FV(M_2)$$

Commentaires :

- L'énoncé comportait une erreur que personne n'a vraiment discuté, la règle de substitution $[x \mapsto M]x'$ sur une variable x' différente de la variable en cours de substitution (x) ne donne évidemment pas z mais x' .
- L'énoncé avait oublié de réclamer également la règle de renommage des variables liées.
- Durant les substitutions, il faut faire attention aux éventuelles captures de variables non souhaitées. Par exemple, le calcul de $[x \mapsto x']\text{let } x' = M_1 \text{ in } M_2$ doit être précédé du renommage de la variable du `let`. Ainsi, cette expression devient $[x \mapsto x']\text{let } y = M_1 \text{ in } [x' \mapsto y]M_2$ si $y \notin FV(M_2)$, on peut alors appliquer la substitution.
- Certains ont répondu $FV(\text{let } x = M_1 \text{ in } M_2) = (FV(M_1) \cup FV(M_2)) \setminus \{x\}$ ce qui dans l'absolu est faux puisque x est lié dans M_1 et donc pas libre. Cette règle peut devenir vraie si l'on impose que les définitions de `let` ne soit pas récursives.

Question 3

- ▷ En étudiant la réduction du terme $\lambda x.\text{let } y = x \text{ in } y$, déterminer à quel terme plus simple, il est équivalent.

On suppose que x et y sont différents. Soit un terme M tel que $y \notin FV(M)$ (si c'est le cas, on peut renommer y dans le `let`) :

$$\begin{aligned} \lambda x.\text{let } y = x \text{ in } y \ M &\rightarrow [x \mapsto M]\text{let } y = x \text{ in } y \\ &\rightarrow \text{let } y = [x \mapsto M]x \text{ in } [x \mapsto M]y \\ &\rightarrow \text{let } y = M \text{ in } y \\ &\rightarrow [y \mapsto M]y \\ &\rightarrow M \end{aligned}$$

Donc, pour tout terme M , on a $\lambda x.\text{let } y = x \text{ in } y \ M \rightarrow M$. Ce `let` est donc équivalent (au sens « se réduisent de la même façon ») au terme $\lambda x.x$ (la fonction identité).

Question 4

Pour typer ML_0 , nous avons besoin de variables de type (notée τ) et des types fonctionnels ($T \rightarrow T$). La grammaire des types est donc :

$$T ::= \tau \mid T \rightarrow T$$

De plus, l'environnement de typage Γ contient l'ensemble des variables définies et leur associe un *schéma de type* (noté S). Un schéma de type est de la forme $\forall \tau_1 \dots \tau_n. T$ où les τ_i sont des variables de type et T est un type.

On définit alors la notion de variable de type libre² par :

$$FTV(\tau) = \{\tau\}$$

$$FTV(T_1 \rightarrow T_2) = FTV(T_1) \cup FTV(T_2)$$

$$FTV(\forall \tau_1 \dots \tau_n. T) = FTV(T) \setminus \{\tau_1, \dots, \tau_n\}$$

$$FTV(\{x_1 : S_1, \dots, x_n : S_n\}) = FTV(S_1) \cup \dots \cup FTV(S_n)$$

De plus, pour simplifier, on suppose que l'ordre de généralisation des variables de type n'est pas important, $\forall \tau. T = T$ si $\tau \notin FTV(T)$ et $\forall. T = T^3$.

Pour passer d'un schéma de type à un type, on définit⁴ la notion d'instanciation par :

$$\forall \tau_1 \dots \tau_n. T' \preceq T \text{ ssi } \exists T_1, \dots, T_n \quad T = [\tau_1 \mapsto T_1, \dots, \tau_n \mapsto T_n]T'$$

Par exemple :

$$\forall \tau. \tau \rightarrow \tau \preceq \tau_1 \rightarrow \tau_1 \quad \forall \tau. \tau \rightarrow \tau \preceq \tau_2 \rightarrow \tau_2 \quad \forall \tau. \tau \rightarrow \tau \preceq (\tau_1 \rightarrow \tau_2) \rightarrow (\tau_1 \rightarrow \tau_2)$$

Une instance d'un schéma est donc un type qui est *inclus* dans ce schéma.

Les règles de typage polymorphe sont alors les suivantes :

$$\frac{\Gamma :: \{x : \tau\} \vdash M : T_M}{\Gamma \vdash \lambda x. M : \tau \rightarrow T_M} \quad (\text{T}_{\text{FUN}}) \quad \frac{\Gamma \vdash M_1 : T \rightarrow T' \quad \Gamma \vdash M_2 : T}{\Gamma \vdash M_1 M_2 : T'} \quad (\text{T}_{\text{APP}})$$

$$\frac{\Gamma(x) \preceq T}{\Gamma \vdash x : T} \quad (\text{T}_{\text{INST}}) \quad \frac{\Gamma \vdash M_1 : T_1 \quad \Gamma :: \{x : \text{Gen}(T_1)\} \vdash M_2 : T_2}{\Gamma \vdash \text{let } x = M_1 \text{ in } M_2 : T_2} \quad (\text{T}_{\text{GEN}})$$

La quatrième règle dite de généralisation est un peu particulière. En effet, pour déterminer le schéma de type que l'on doit associer à x pour typer M_2 , il faut :

1. typer M_1 (ce qui permet de déterminer la valeur de T_1),
2. généraliser ce type, *i.e.* généraliser toutes ses variables libres (sous forme mathématique $\text{Gen}(T) = \forall \tau_1 \dots \tau_n. T$ avec $\{\tau_1, \dots, \tau_n\} = FTV(T)$)

²l'intuition est la même que pour les variables libres.

³On peut donc ajouter un type dans l'environnement ...

⁴La notion de substitution est la même que pour les variables et laissée à votre sagacité!

- ▷ Produisez la dérivation de typage du terme $\lambda x.\text{let } y = x \text{ in } y$. Que pensez-vous de la correction de ce système de type au vue de la réponse à la question précédente ? Expliquer.

$$\frac{\frac{\frac{}{\{x : \tau\} \vdash x : \tau} \quad \frac{\frac{}{\{x : \tau, y : \forall \tau. \tau\} \vdash y : \tau'}}{\forall \tau. \tau \preceq \tau'}}{\{x : \tau\} \vdash \text{let } y = x \text{ in } y : \tau'}}{\{\} \vdash \lambda x. \text{let } y = x \text{ in } y : \tau \rightarrow \tau'}}$$

Or, d'après la question précédente, $\lambda x.\text{let } y = x \text{ in } y \equiv \lambda x.x$ dont le type est $\forall \tau. \tau \rightarrow \tau$ et $\forall \tau. \tau \rightarrow \tau \not\preceq \tau \rightarrow \tau'$ si $\tau \neq \tau'$ ⁵. On obtient donc une contradiction entre équivalence et typage. Le système de type présenté n'est donc pas sûr !

En fait, il ne faut pas généraliser toutes les variables de type libre dans l'opération *Gen*. Consultez la référence cité en fin de problème pour en savoir plus.

Question 5

Il est possible de faire une analogie entre le polymorphisme de généralisation décrit ci-dessus et le polymorphisme de la généricité dans les langages objets (une classe correspondant à une fonction par exemple).

- ▷ Quelle différence y a t'il entre le système de ML_0 et la généricité des langages objets (vous pouvez vous inspirer de l'exercice 3) ?

Dans la généricité des langages objets (et donc celle de Java 1.5 décrite dans l'exercice 3, les types à généraliser sont donnés de manière explicite lors de la définition d'une classe générique. À l'opposé dans le λ -calcul (et dans les langages qui en sont directement issus comme ocaml par exemple), ces types ne sont pas donnés et doivent être calculé par le système de type.

Remarque :

Il existe évidemment une solution au problème de la généralisation dans les langages fonctionnels. Si vous êtes intéressés, vous pourrez consulter par exemple la thèse de Xavier Leroy : « Typage polymorphe d'un langage algorithmique ». Thèse de doctorat, Université Paris 7, 1992.

⁵Dans la dérivation précédente, on peut évidemment toujours choisir τ' différent de τ .