Spécialisation et sous-typage : thème et variations

Roland Ducournau

L.I.R.M.M. – 161, rue Ada
34392 Montpellier cedex 5
ducournau@lirmm.fr

RÉSUMÉ. *La spécialisation de classes constitue la caractéristique la plus remarquable de l'approche objet mais son assimilation au sous-typage, dans le cadre de la programmation par objets, conduit à la problématique bien connue du caractère covariant ou contravariant de la redéfinition. Pour que le typage soit* sûr, *le sous-typage impose une redéfinition* contravariante *contraire à la sémantique* covariante *de la spécialisation. Ce problème est bien connu, mais le point de vue du typage sûr est resté dominant. Cet article a pour objet d'analyser précisément le problème, de montrer l'irréductibilité du conflit, et la* nécessité *du risque d'erreur de type. La plupart des langages reconnaissent d'ailleurs implicitement cette nécessité en permettant une coercition de types contraire à la sûreté du typage. Nous montrons aussi que les nombreuses alternatives comme la sélection multiple, la généricité ou le filtrage restent illusoires ou insuffisantes pour traiter le problème. La solution réaliste est alors d'adopter la redéfinition covariante en rajoutant aux langages les éléments permettant de circonscrire précisément les erreurs de type, et en complétant les méthodes d'analyse et de conception pour qu'elles les prennent en compte.*

ABSTRACT. *Class specialization is the most original feature of object orientation, but identifying it to subtyping leads to the well known covariance-contravariance controversy. Type safety requires contravariance while specialization needs covariance. This paper aims to precisely analyse this problem, to show how irreductible it is and the need for type errors. We show that many alternatives as multiple dispatch, genericity or matching cannot solve the problem. Thus, a realistic solution is to adopt covariant redefinition as the basis of a type system where type errors are explicited, as they should be at the analysis and design stages.*

MOTS-CLÉS : *langage à objets, spécialisation, sous-typage, généricité, héritage, sélection multiple, covariance, contravariance.*

KEYWORDS: *object-oriented languages, specialization, subtyping, genericity, inheritance, multiple dispatch, covariance, contravariance.*

1. Introduction

En vingt ans, le modèle objet s'est imposé, avec une certaine hégémonie, à toutes les activités du génie logiciel. Ce succès peut s'expliquer par sa double adéquation :

– avec le modèle de représentation du monde de la pensée occidentale, tel qu'il a été codifié par les traditions aristotélicienne et platonicienne [WEG 86, WAN 89, PER 98, RAY 00], ce qui fait du modèle objet le paradigme de modélisation idéal dans la phase d'analyse, aussi bien qu'en représentation des connaissances ;

– avec les principales exigences de qualité du génie logiciel, en particulier la modularité, l'extensibilité, la réutilisabilité et la fiabilité.

La domination de l'approche objet sur toutes les phases du développement logiciel repose d'abord sur la continuité du modèle qui peut être assurée depuis l'analyse la plus intuitive et « naturelle » jusqu'à l'implémentation la plus technique.

La spécialisation de classes constitue certainement l'aspect le plus remarquable de l'approche objet, à la fois origine de ces principales qualités et rupture avec les paradigmes précédents de programmation. La littérature, que ce soit en matière de programmation ou d'analyse et de conception, lui accorde ainsi une place éminente : voir, par exemple, [BOO 94, chapitre 4] ou [WEG 86]. Il s'agit probablement du thème le plus largement représenté, sous des catégories, problématiques et terminologies variées : classification, spécialisation, héritage, délégation, sous-typage, envoi de message, etc. Nous posons donc, en pétition de principe, que la modélisation objet résultant de la phase d'analyse possède une sémantique de spécialisation, à la Aristote, et nous considérons la question de la mise en œuvre, dans un langage de programmation par objets, d'une spécification relevant de cette sémantique de spécialisation.

Or cette continuité des modèles ne va pas sans difficultés : elle se heurte à un obstacle aussi subtil qu'irréductible, le fameux conflit entre *covariance* et *contravariance* qui divise la communauté de la programmation par objets — voir, par exemple, [CAS 95], [MEY 97, chapitre 17] ou [SZY 94] — et qui est à la source de diverses tentatives pour distinguer la spécialisation du sous-typage [COO 90, OMO 95]. L'objectif de cet article est de faire le point sur ce débat, d'en expliquer les causes profondes et les conséquences inéluctables, de vérifier le caractère illusoire de certaines alternatives et de proposer un compromis *objectif* qui tienne compte du fait que les langages de programmation doivent être des serviteurs des modèles d'analyse et de conception, et non l'inverse. La sûreté du typage reste pour nous un mythe. Les erreurs de type sont inhérentes à la spécialisation : il faut donc les prendre en compte dès l'analyse et la conception et concevoir des langages qui assurent le meilleur compromis entre la *sûreté* et l'expressivité, c'est-à-dire sa proximité de la sémantique naturelle de la spécialisation. Cet article n'a la prétention d'apporter aucune théorie nouvelle ni même un point de vue véritablement original. Les éléments que nous allons confronter sont connus mais il ne semble pas que la communauté objet — en particulier les concepteurs de langages et les théoriciens des types — en ait tiré toutes les conséquences. D'une certaine manière, nous reprenons l'analyse de [MEY 97, chapitre 17], en en acceptant l'essentiel des prémisses et une grande partie des analyses, mais en divergeant au niveau des conclusions.

Le plan de l'article est le suivant. La section 2 expose les éléments du problème, avec la présentation de ce qui pourrait être appelé le « modèle standard » de l'approche objet, suivie par celle de la spécialisation conceptuelle et de son exigence de covariance, pour finir par le sous-typage et la règle de contravariance. Les raisons du conflit sont examinées. La section suivante décrit deux variations importantes autour de la sélection de méthodes : la *surcharge statique* à la C++ ou JAVA et la sélection multiple à la CLOS. La section 4 montre ensuite l'inéluctabilité des erreurs de type, en analysant comment les langages standards sont condamnés à simuler la covariance ou le typage sûr. La section 5 présente alors des alternatives : la généricité et la distinction entre classes et types ou entre spécialisation et sous-typage. La section 6 considère ensuite le caractère plus ou moins global ou séparé de la compilation et son influence sur le typage. La section 7 propose enfin le compromis qui nous semble préférable : il assure la continuité des modèles, au prix d'un risque, inévitable mais objectivement encadré, d'erreurs de types. Quelques perspectives viennent en conclusion.

2. Covariance et contravariance

Le problème réside dans l'imperfection du recouvrement entre trois versions voisines de la spécialisation : la spécialisation de classes du modèle objet standard, la spécialisation conceptuelle héritée de la logique la plus antique, et la spécialisation de types, ou sous-typage, liée au polymorphisme dit d'inclusion.

2.1. *Le modèle standard*

Le modèle objet standard est le modèle de langages dits de classes, selon [MAS 89]. Il est constitué :

– d'une hiérarchie de spécialisation de classes : on parle aussi de hiérarchie d'héritage, cet héritage pouvant être simple ou multiple suivant qu'une classe peut avoir une unique super-classe directe ou plusieurs ;

– chaque classe décrit des propriétés, attributs et méthodes, qui s'ajoutent aux propriétés qu'elle hérite de ses super-classes, ou qu'elles remplacent : c'est l'*intension* de la classe ; pour simplifier, nous considérerons que toutes les propriétés sont *publiques*, c'est-à-dire qu'elles sont visibles et utilisables partout ;

– des objets, instances de ces classes, sont créés par un procédé d'instanciation d'une classe : c'est l'*extension* de la classe ;

– enfin, l'activation des méthodes attachées aux classes s'effectue suivant la métaphore de l'*envoi de message*, appelée aussi *liaison tardive*.

Les langages relevant de ce modèle standard se différencient suivant que leur typage est purement dynamique — comme SMALLTALK [GOL 83] ou CLOS [BOB 88] — ou statique — comme C++ [STR 98], EIFFEL [MEY 92, MEY 97] ou JAVA [ARN 97]. Entre ces deux extrêmes, toute la gamme des intermédiaires est possible : par exemple CLAIRE, un langage avec un typage statique « faible » puisqu'il autorise

un typage aussi bien statique que dynamique [CAS 99], ou OBJECTIVE-CAML, qui étend aux objets l'inférence de types de ML [RéM 97]. Cet article n'est évidemment concerné que par le typage statique, et plus précisément par les langages au typage dit fort parce qu'il tend à exclure toute erreur de type à l'exécution.

2.2. *La covariance de la spécialisation conceptuelle*

La sémantique de la spécialisation — que l'on qualifiera de conceptuelle pour la distinguer de ses avatars informatiques — remonte au moins à Aristote et s'illustre par le syllogisme classique : *Socrate est un homme, les hommes sont mortels, donc Socrate est mortel*. Nous postulons que la spécialisation de classes de l'approche objet est une spécialisation conceptuelle, ce qui nous autorise à traduire le syllogisme dans le vocabulaire objet. Donc, *Socrate* est une instance, *homme* et *mortel* sont des classes.

2.2.1. *Inclusion des extensions*

En accord avec la tradition aristotélicienne, on généralise en disant que *les instances d'une classe sont des instances de ses super-classes*. Plus formellement, on introduit la relation de spécialisation $\prec$ (B sous-classe de A s'écrira $B \prec A$) et une fonction Ext qui associe à chaque classe son *extension*, l'ensemble de ses instances :

$$B \prec A \Longrightarrow Ext(B) \subseteq Ext(A) \quad (1)$$

Cette inclusion des extensions constitue l'essence de la spécialisation. Elle a deux conséquences « naturelles » : une inclusion en sens inverse des intensions (c'est l'héritage), et une inclusion des domaines des propriétés (c'est la redéfinition covariante).

2.2.2. *Héritage des propriétés*

La logique classique continue à s'appliquer lorsque l'on considère les propriétés des classes : ce sont en fait les propriétés des instances, mais leur description est factorisée dans les classes. Si B est une sous-classe de A, les instances de B, étant des instances de A, ont toutes les propriétés des instances de A. On dit donc que B hérite des propriétés de A. Plus formellement, on introduit une fonction Int qui associe à chaque classe son *intension*, c'est-à-dire l'ensemble de ses propriétés :

$$B \prec A \Longrightarrow Int(A) \subseteq Int(B) \quad (2)$$

L'inclusion des intensions se fait en sens inverse de l'inclusion des extensions. Ces inclusions en sens contraires des extensions et des intensions sont à la base des treillis de Galois qui fondent les approches de calcul automatique de hiérarchies de classes [GOD 95] et qui sont aussi dénommés « treillis de concepts » [WIL 92].

2.2.3. *Spécialisation des domaines*

Ces propriétés sont valuées dans les objets et peuvent être décrites dans les classes par un *domaine*, c'est-à-dire un ensemble de valeurs possibles : la *valeur* de la pro-

priété dans une instance de la classe est contrainte à appartenir au *domaine* de la propriété dans la classe. Ainsi, par exemple, les `Personnes` ont un `âge`, dont le domaine est l'intervalle entier $[0, 120]$. Lorsque l'on spécialise une classe, on spécialise aussi le domaine des propriétés de la classe :

$$B \prec A \ \& \ P \in Int(A) \Longrightarrow Dom(B, p) \subseteq Dom(A, p) \tag{3}$$

L'`âge` des `Enfants`, spécialisation de `Personnes`, a pour domaine $[0, 16]$. Cet exemple concerne un attribut, mais on trouve facilement l'analogue pour une méthode : un `Animal` `mange` `N'importe-quoi` , un `Carnivore` `mange` des `Animaux`, et un `Phoque` `mange` des `Poissons`, etc.

Si l'on définit le domaine comme l'ensemble des valeurs effectives prises par la propriété dans l'extension de la classe, l'inclusion (3) est une conséquence logique de (1). Bien qu'intuitif, ce résultat nécessite une formalisation de la sémantique de la spécialisation et de l'extension des classes, vue comme l'ensemble des instances *potentielles* de la classe, qui dépasse le présent article : pour une étude plus approfondie de cette sémantique de la spécialisation et de ses conséquences, le lecteur est renvoyé à [DUC 96, DUC 98, DUC 01b] et à toute la littérature sur les logiques de descriptions [WOO 92]. En revanche, si le domaine n'est qu'une borne supérieure de l'ensemble de ces valeurs effectives, l'inclusion (3) n'est plus qu'une convention. A l'extrême, le domaine ainsi spécialisé devient vide : [RUM 91] donne en exemple le cas d'un `cercle`, sous-classe d'`ellipse`, dont les méthodes de déformation suivant l'axe des x ou des y doivent être bloquées, puisqu'elles feraient perdre au `cercle` sa caractéristique. Nous n'examinerons pas cette généralisation qui reviendrait à admettre des exceptions à l'inclusion (2) mais sa prise en compte serait analogue à celle de la redéfinition des domaines.

2.2.4. *Covariance et contravariance dans la spécialisation*

La spécialisation est ainsi une structure d'ordre à chaque élément de laquelle peuvent être rattachées diverses entités qui évoluent de façon monotone relativement à la relation d'ordre. Certaines sont *covariantes* car elle évoluent dans le même sens : c'est le cas de l'extension et du domaine, et c'est ainsi qu'il faut comprendre la « covariance de la spécialisation ». D'autres sont *contravariantes*, car elles évoluent en sens inverse : c'est le cas des intensions. On va voir que le typage introduit des entités dont l'évolution est contravariante alors que celle de leur correspondant le plus proche dans la spécialisation est covariante.

2.3. ***La contravariance du sous-typage***

2.3.1. *La conformité entre types*

La définition, presque naïve, de la correction du typage se formule de la façon suivante : une variable ne peut être liée à une valeur que si le type de la valeur « se conforme » au type de la variable. Cette règle vaut autant pour les affectations que

pour le passage de paramètres. Le typage repose ainsi sur une relation de *conformité* (en anglais, *conformance*) entre types. Cette relation doit être un préordre (transitive et réflexive) et elle se réduit à l'identité, en l'absence de polymorphisme. Le polymorphisme traduit ainsi une relation de conformité non triviale : il faut alors distinguer *type statique* (celui qui annote les expressions du programme) et *type dynamique* (celui des valeurs à l'exécution, c'est-à-dire la classe qui les a instanciées). La règle précédente assure qu'à l'exécution, le type dynamique d'une entité se conformera toujours à son type statique. La *liaison tardive* consiste à faire dépendre le code exécuté du type dynamique de l'argument (le receveur). La règle qui concerne l'envoi de message se précise ainsi : le type du receveur doit connaître le message, c'est-à-dire qu'il doit avoir une méthode de ce nom.

La façon la plus simple de faire respecter ces règles est de les imposer sur les types statiques. Dans une affectation ou un passage de paramètre, le type *statique* de la source doit se conformer à celui de la cible. Dans un envoi de message, le type *statique* du receveur doit connaître le message. C'est une simplification : elle n'est ni nécessaire — la *correction dynamique* d'un programme, pour toutes ses exécutions, n'impose pas une telle *correction statique*, pour toutes ses expressions — ni suffisante — cela dépend de la relation de conformité. Nous nous placerons pour l'essentiel dans ce cadre mais il n'est pas universel : certains systèmes de types n'imposent pas cette contrainte sur les types statiques, préférant une plus grande flexibilité grâce à des inférences de types ou au prix de vérifications dynamiques [JOS 01, ROY 02].

2.3.2. *Types, classes et domaines*

La spécialisation de classes conduit à ce que l'on appelle aussi bien polymorphisme d'inclusion que sous-typage. Un type est souvent vu comme un ensemble de valeurs et un ensemble d'opérateurs qui s'appliquent à ces valeurs, bref comme une extension et une intension. Il est donc assez naturel de chercher à faire des classes des types, ou, de façon à peu près équivalente, de chercher à associer un type à chaque classe. Le type — éventuellement exprimé directement par la classe — va alors servir à annoter les éléments du programme, les simples identificateurs, mais aussi les propriétés des classes, dont le *domaine* sera exprimé par un *type*. Il est tout aussi naturel de faire coïncider la spécialisation avec une notion de *sous-typage* : intuitivement, un sous-type serait défini par un sous-ensemble des valeurs du type et un sur-ensemble de ses opérateurs, un peu comme une sous-classe. Le sous-typage, que l'on notera $<:$, est ainsi une spécialisation, puisqu'il repose sur une inclusion des extensions : *an instance of a subtype can always be used in any context in which an instance of a type was expected* [WEG 88]. Cette caractérisation du sous-typage comme une spécialisation se fait souvent au travers d'une règle dite de subsomption [CAS 96b] :

$$\frac{u <: v \qquad x : u}{x : v} \tag{4}$$

qui exprime, pour l'essentiel, la même chose que (1) : si x a le type u et que u est un sous-type de v, alors x a le type v.

Identifier le sous-typage à la spécialisation de classes revient à faire commuter le diagramme suivant :

$$\begin{array}{ccc} Classes & \xrightarrow{type} & Types \\ {\scriptstyle \prec}\downarrow & & \downarrow{\scriptstyle <:} \\ Classes & \xrightarrow[type]{} & Types \end{array} \tag{5}$$

ce que l'on peut aussi exprimer ainsi :

$$B \prec A \Longrightarrow type(B) <: type(A) \tag{6}$$

Mais c'est là que tout se complique.

2.3.3. *Rôle du typage*

Pour définir le sous-typage de façon satisfaisante, il faut revenir sur le rôle du typage, lorsqu'il est statique. A quoi sert-il ? Essentiellement à trois choses :

– à permettre au compilateur d'assurer qu'il n'y aura pas d'erreur de type à l'exécution, c'est-à-dire soit l'application d'un opérateur à une valeur d'un type qui ne supporte pas l'opérateur — c'est l'erreur `"message inconnu"` —, soit le passage d'un argument dont le type ne se conforme pas à celui du paramètre — c'est l'erreur `"mauvais type d'argument"` ;

– à permettre au compilateur de lier, le plus statiquement possible pour des raisons d'efficacité, les noms de méthodes ou procédures au code à exécuter ;

– enfin, à auto-documenter les programmes, les annotations de type servant de commentaires ou de spécification explicite [MEY 97].

C'est l'absence d'erreur de types qui nous concernera en premier lieu. L'idéal du type statique est de se faire éliminer par le compilateur, qui prouve la correction du programme tout en produisant un code dans lequel toute référence aux types statiques a disparu. Si la correction n'est pas prouvée, des erreurs de type résiduelles sont possibles, qui nécessiteront des vérifications dynamiques à l'exécution ou qui entraîneront un comportement imprévisible des programmes. Notons que cet idéal de l'élimination des types peut être incompatible avec certaines implémentations, indépendamment de la correction du système de types considéré : l'implémentation de l'héritage multiple en compilation séparée est à cet égard caractéristique [ELL 90, DUC 01a].

2.3.4. *Sous-typage et substituabilité.*

Le compilateur doit assurer cette absence d'erreur pour toute exécution possible, ce qui peut s'approximer en définissant le sous-typage par la notion de *substituabilité*. Un type t_1 est un sous-type d'un type t_2 si toute valeur du type (dynamique) t_1 peut être substituée, à l'exécution, à toute expression du type (statique) t_2, *sans déclencher d'erreur de type*. Cette formulation précise le « *can always* » de la citation précédente [WEG 88] pour définir ce qu'est un sous-typage *sûr*. Ainsi, pour qu'une sous-classe puisse être considérée comme un sous-type, il faut que la définition de la sous-classe

relativement à la super-classe respecte les contraintes imposées par la substituabilité. Inversement, une classe peut être un sous-type sans être une sous-classe : deux classes de même définition (au nom près) seront parfaitement substituables. De fait, si la relation de spécialisation est un ordre partiel, la relation de sous-typage, au sens strict de la substituabilité, n'est qu'un préordre. En pratique, les langages qui identifient types et classes prennent la relation de spécialisation comme relation de sous-typage et les substituabilités qui ne coïncident pas avec la spécialisation ne sont pas considérées.

Le principe de substituabilité constitue une approche essentiellement pessimiste [MEY 97] : comme pour la complexité, on considère « le pire des cas ». Tout dépend alors de la dimension du contexte — c'est-à-dire de l'unité de programme, l'expression, la méthode, la classe, le module ou la totalité du programme — que l'on s'autorise à examiner pour déterminer ce pire des cas : le pessimisme est une fonction décroissante du contexte. L'objectif d'une théorie du typage purement statique serait donc de définir la relation de conformité la plus large possible qui fasse que la définition naïve du typage qui précède assure une parfaite substituabilité. Si la conformité a de bonnes propriétés de localité, le typage résultant peut permettre une compilation séparée (cf. section 6).

2.3.5. *Substituabilité et redéfinition.*

Pour qu'une sous-classe puisse être considérée comme un sous-type, il est nécessaire de contraindre la façon dont un attribut ou une méthode, définis dans une classe, peuvent être redéfinis dans une sous-classe. En effet, l'introduction dans la sous-classe de nouvelles propriétés — c'est-à-dire de propriétés non définies dans les super-classes, ce que [WEG 88] appelle *extension horizontale* — ne pose aucun problème. En revanche, la redéfinition — *modification verticale* pour [WEG 88] — peut en poser lorsqu'elle concerne le type d'un attribut ou la signature d'une méthode, c'est-à-dire les types des paramètres et le type de retour.

NOTATIONS. — On note $m_A(u) : v$ une méthode de nom m, définie dans la classe A avec un type de paramètre u et un type de retour v. Le cas de plusieurs paramètres se ramène à un paramètre unique dans le produit des types (cf. section 3.2). La notation met bien en évidence la dissymétrie entre le receveur, sur lequel repose la sélection, et les paramètres secondaires, qui n'y ont aucune part. En cas d'absence de paramètres, on la note $m_A() : v$. Pour un attribut, on utilise la notation $m_A : v$. Suivant le contexte, les annotations de type ou de classe pourront être omises. Enfin, pour les types statique et dynamique du receveur, on utilise les notations respectives τ_s et τ_d, avec $\tau_d <: \tau_s$.

Redéfinition covariante du type de retour.

Soit une méthode de nom m définie dans la classe A avec un type de retour t — notée $m_A() : t$ — et redéfinie dans la classe B, sous-classe de A, avec un type de retour u — notée $m_B() : u$. Dans tout contexte comme :

```
{ x : A;   y : t;  y := x.m();}
```

x doit être substituable par toute instance de A, donc en particulier de B, ce qui impose que u soit un sous-type de t. La redéfinition est *covariante* sur le type de retour, puisque les types varient dans le même sens.

Redéfinition contravariante du type des paramètres.

Soit une méthode de nom m définie dans la classe A avec un type de paramètre t — notée $m_A(t)$ — et redéfinie dans la classe B, sous-classe de A, avec un type de paramètre u — notée $m_B(u)$. Dans tout contexte comme :

```
{ x : A;   y : t;   x.m(y);}
```

x doit être substituable par toute instance de A, en particulier par une instance de B, et la méthode m de A doit donc accepter toute valeur du type t comme paramètre, ce qui impose que t soit un sous-type de u. En particulier, les deux méthodes doivent avoir le même nombre de paramètres. La redéfinition est *contravariante* sur le type du paramètre, puisque les types varient en sens contraire. C'est la *règle de contravariance*.

Redéfinition invariante du type d'un attribut.

Le cas d'un attribut $m_A : t$ se ramène à celui d'une méthode :

– lorsqu'on accède en lecture à l'attribut, on peut le considérer comme une méthode sans paramètre, $m_A() : t$, dont le type de retour est celui de l'attribut : la redéfinition doit être covariante ;

– lorsqu'on accède en écriture (affectation) à l'attribut, on peut le considérer comme une méthode, $m_A(t)$, avec un paramètre, la valeur à affecter, dont le type est celui de l'attribut : la redéfinition doit être contravariante.

On en conclut qu'il n'est pas possible de redéfinir le type d'un attribut dans une sous-classe : la redéfinition est *invariante*.

Attributs immutables et types fonctionnels

Cette dernière conclusion ne vaut que pour les attributs *mutables*, c'est-à-dire dont on peut modifier la valeur. Lorsque ce n'est pas le cas — l'attribut est dit *immutable* —, sa redéfinition est covariante. [WEG 88] distingue ainsi une spécialisation « en lecture seule », qui serait celle d'un modèle objet purement fonctionnel : mais un tel modèle perdrait l'essentiel du modèle objet, l'identité des objets. La relation entre la règle de contravariance et les types fonctionnels se retrouve comme suit. Une méthode $m(t_1) : u_1$ peut être considérée comme un attribut immutable dont le type est fonctionnel, noté $m : t_1 \rightarrow u_1$. Elle peut donc être substituée (redéfinie) par une méthode de type $t_2 \rightarrow u_2$, sous-type de $t_1 \rightarrow u_1$. Or le sous-typage des types fonctionnels impose que

$$t_2 \rightarrow u_2 <: t_1 \rightarrow u_1 \iff t_1 <: t_2 \;\;\&\;\; u_2 <: u_1 \tag{7}$$

Pour être substituable, le sous-type fonctionnel doit accepter plus d'arguments et retourner moins de valeurs. On constate ainsi que, contrairement à une opinion erronée[1], le typage des objets a besoin d'un équivalent d'un typage fonctionnel.

Types paramétrés

La règle de contravariance s'étend aussi aux types ou classes paramétrés, ce que l'on appelle en C++ des *templates*. Soit le type paramétré $\texttt{set}\langle T\rangle$ des ensembles dont

1. *In type theory,* contravariance *is used instead* [of covariance]. *Contravariance is the proper notion when functions are passed as arguments, which is not the case here* [ABI 95, page 553].

les éléments sont d'un type représenté par le type formel T, une classe A et sa sous-classe B. Alors, si $\texttt{set}\langle B\rangle$ est manifestement une spécialisation de $\texttt{set}\langle A\rangle$ — par l'inclusion de leurs extensions —, ce n'en est pas un sous-type. Considérons la méthode d'ajout d'élément, dont le paramètre est de type formel T, $\texttt{addelt}_{\texttt{set}\langle T\rangle}(T)$. L'instanciation de ce type formel par les types A ou B, donnera respectivement les méthodes $\texttt{addelt}_{\texttt{set}\langle A\rangle}(A)$ et $\texttt{addelt}_{\texttt{set}\langle B\rangle}(B)$: s'il y avait sous-typage, la redéfinition serait strictement covariante. La règle de contravariance ne s'applique qu'aux occurrences du type formel en position contravariante (paramètre ou attribut). En leur absence, le sous-typage de $A\langle B\rangle$ par $A\langle C\rangle$ reste sûr [COO 89, WEB 92].

REMARQUE. — Les termes de covariance et contravariance ont été introduits dans ce cadre par [CAR 84] à propos des types fonctionnels. Comme la covariance du type de retour ne pose aucun problème — si ce n'est d'implémentation efficace [DUC 01a] —, nous utiliserons les termes de covariance et de contravariance pour désigner la redéfinition du type des paramètres ou des attributs. Par extension, on pourra aussi qualifier de contravariant ou covariant des entités variées, suivant qu'elles respectent, ou non, la règle de contravariance (langage, typage, héritage, paramètre, etc.) ou suivant qu'elles sont en position de type de paramètre ou de type de retour (types formels).

2.4. *L'irréductibilité du conflit*

2.4.1. *La cause du conflit*

La section précédente a présenté les éléments du conflit, qui se ramène à une incompatibilité entre les trois exigences de la spécialisation conceptuelle, qui implique une covariance des domaines, du polymorphisme d'inclusion et de la garantie d'absence d'erreurs de types à l'exécution. Ces trois termes sont deux à deux compatibles : on peut avoir covariance et polymorphisme, covariance sans erreurs de types, et polymorphisme sans erreurs de types.

La cause de ce conflit est simple, même si elle n'a sans doute jamais été précisément énoncée. Elle réside dans l'identification abusive du *domaine* et du *type* des propriétés ou, plus exactement, dans la traduction du premier par le second. Or leurs sémantiques respectives diffèrent sur un point fondamental : la première est fondée sur une quantification existentielle — il existe une valeur du domaine qui value la propriété, un `phoque` `mange` *un* `poisson` —, alors que la seconde est fondée sur une quantification universelle — on doit pouvoir substituer *n'importe quel* `poisson` comme argument de la méthode `manger`. La règle (7) constitue une spécialisation, à condition de bien la formuler : une fonction totale dont le domaine (au sens mathématique) contient t_2 est bien une fonction dont le domaine contient t_1, lorsque $t_1 <: t_2$. Mais la spécialisation est inversée si l'on considère des fonctions partielles, dont le domaine est respectivement inclus dans t_1 et t_2.

2.4.2. *Continuité des modèles*

Notre hypothèse de départ est la suivante : 1) les méthodes d'analyse produisent un modèle relevant, pour l'essentiel, de la spécialisation conceptuelle ; 2) ce modèle

devrait se projeter sur un modèle de conception et d'implémentation basé sur une hiérarchie de classes reproduisant — de la façon la plus directe et la plus exacte, à la limite de l'isomorphisme — la spécialisation conceptuelle d'origine ; 3) l'implémentation doit pouvoir se baser sur le polymorphisme, qu'il soit d'inclusion (sous-typage) ou paramétrique (généricité) pour minimiser les redondances et maximiser la factorisation, tout en étant fidèle à la spécialisation sous-jacente.

Nous ne prétendons pas que tout programme objet doive se conformer à ces principes mais, plus simplement, que tout langage doit permettre de s'y conformer. En particulier, nous revendiquons le fait qu'une hiérarchie de classes doive refléter une sémantique de spécialisation et non pas, comme c'est malheureusement trop souvent le cas, une sémantique d'agrégation [CAR 93] : la règle 8 de [JOH 88] (*Subclasses should be specializations*) et non pas l'exemple aberrant de [BOB 88], où une `tarte-aux-pommes` est sous-classe de `pâte`, `pomme` et `cannelle`. Dans l'ensemble, les méthodes d'analyse et de conception soulignent cette sémantique de spécialisation de l'héritage, quitte à proposer des solutions de conception agrégatives *ad hoc* pour traiter des cas particuliers [BOO 94, RUM 91].

Si la découverte de la règle de contravariance dans le sous-typage est assez ancienne, il revient à [CAS 95] d'avoir fait l'analyse théorique de ce conflit. Mais ses conclusions — prendre la covariance si l'on privilégie la spécialisation, la contravariance si l'on préfère la substituabilité — ne permettent pas de dépasser le conflit. Le problème est qu'il n'est pas possible de préférer l'une ou l'autre : il faudrait les deux, à divers moments du cycle de vie. L'analyse va privilégier la spécialisation, alors que l'implémentation est seule concernée par la substituabilité.

Il faut donc considérer comme illusoire, soit la continuité des modèles de l'analyse à l'implémentation, soit la possibilité d'éviter toute erreur de type. Le pire pouvant être d'avoir simultanément discontinuité des modèles *et* erreurs de type, ce qui n'est malheureusement pas exclu.

2.4.3. *La variété des réponses*

En pratique, les langages ont donné quatre catégories de réponse à ce problème.

– Quelques langages comme EIFFEL ou O_2 autorisent une rédéfinition covariante, en essayant de contraindre le langage pour empêcher les erreurs de type.

– Les langages C++ ou JAVA privilégient la sûreté du typage et, sans doute, la simplicité de l'implémentation : la règle de contravariance est même renforcée puisque le type des paramètres est invariant (personne n'ayant jamais trouvé l'usage d'une redéfinition véritablement contravariante). JAVA présente en outre deux curiosités : le type de retour est lui aussi invariant, ce qui n'a aucune justification, et le sous-typage des tableaux est covariant — si $B \prec A$, $B[\,]$ est un sous-type de $A[\,]$.

– Le problème résidant dans les paramètres secondaires non concernés par la sélection dynamique, la sélection multiple de CLOS le contourne en faisant participer tous les paramètres à la sélection.

– Enfin, divers langages proposent des solutions basées sur un découplage entre classes et types, spécialisation et sous-typage. En SATHER, le sous-typage respecte la règle de contravariance, et l'héritage permet des redéfinitions covariantes [OMO 95]. POLYTOIL et LOOM proposent une relation entre types, le filtrage (*matching*), plus faible que le sous-typage [BRU 95, BRU 97] et limitant le polymorphisme.

Les sections suivantes analysent plus en détail la façon dont le problème est affronté ou contourné par ces différentes approches.

3. La sélection de méthodes

La présentation du problème a été faite dans le cadre idéal des abstractions les plus caractéristiques de l'approche objet. Une analyse plus approfondie nécessite de discuter de la sélection dynamique de méthodes en prenant en compte, soit des variantes (sélection multiple), soit des modes de sélection préexistant historiquement à l'approche objet et qui ont malheureusement persisté (surcharge statique).

3.1. *La surcharge statique : cas de* C++ *et* JAVA

De nombreux langages comme C++, JAVA ou SATHER proposent, au prix d'une confusion importante, un second mode de sélection, la *surcharge statique*.

3.1.1. *Surcharges statique et dynamique*

Dans le contexte des langages de programmation, la notion de polymorphisme est un peu ambigüe. Elle s'applique aux types, en désignant le fait qu'un même type abstrait soit implémenté par des classes différentes. Elle désigne ainsi la capacité qu'a une forme syntaxique de pouvoir être liée à des valeurs de types différents. Mais elle désigne aussi, improprement puisqu'il s'agit plutôt de polysémie, la capacité qu'a une forme syntaxique de traduire différents comportements suivant le contexte. Dans ce dernier sens, le polymorphisme se ramène, pour la plus grande part, à la notion de *surcharge* (ou *overloading*) qui désigne le fait qu'un même nom soit utilisé dans des sens différents. La surcharge doit donc se résoudre par un mécanisme de sélection ou de désambiguïsation, qui permet d'en distinguer deux grandes catégories suivant que cette sélection se fait statiquement, à la compilation, ou dynamiquement, à l'exécution.

L'envoi de message, la liaison tardive et ce que nous avons appelé *redéfinition*, relèvent de la *surcharge dynamique*. C++ et JAVA — à la suite de PL/1 ou C — autorisent aussi une *surcharge statique* des méthodes : dans ces deux langages, la redéfinition étant invariante sur le type des paramètres, la définition de méthodes, de même nom mais différant par les types des paramètres, est autorisée et interprétée comme une surcharge statique. La surcharge statique ne pourrait pas coexister avec une redéfinition non invariante, sans de grandes difficultés, d'où son exclusion d'EIFFEL. Dans la classification du polymorphisme proposée par [CAR 85], la surcharge statique est du polymorphisme *ad hoc*, alors que les implémentations différentes d'un

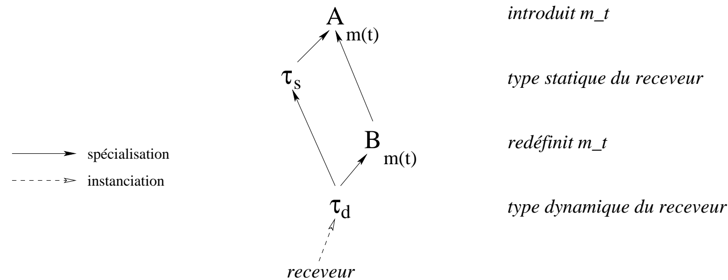

Figure 1 – Surcharge statique et dynamique : les 4 classes impliquées dans la sélection.

même type relèvent du polymorphisme universel, qui se décline lui-même en polymorphisme d'inclusion (sous-typage) ou polymorphisme paramétrique (généricité).

3.1.2. *Surcharge statique, entités génériques et renommage*

Une analyse « conceptuelle » de la spécialisation et de l'héritage conduit à considérer que le nom des propriétés, telles qu'elles sont définies, héritées et utilisées, désigne une « entité générique » sous-jacente, invariante par héritage et redéfinition. C'est le point de vue défendu dans [DUC 95] à propos du *conflit de nom* et illustré par CLOS avec la distinction entre *fonction générique* et *méthode*. Ce sont ces entités génériques qui constituent l'*intension* des classes. En vertu de la sémantique de la spécialisation énoncée plus haut (section 2.2), elles sont héritées de façon monotone : toutes les méthodes de A de nom m sont accessibles dans sa sous-classe B, c'est-à-dire sur une variable de type statique B. Dans cette vision conceptuelle des objets, la surcharge statique est un phénomène d'ambiguïté du nommage des entités génériques, alors que la surcharge dynamique, l'envoi de message et l'héritage sont des phénomènes relatifs à la même entité générique, dont les différentes occurrences (méthodes par exemple) peuvent se masquer et sont l'objet d'une sélection. Dans cette optique, il est donc possible de renommer les méthodes — par exemple en concaténant à leur nom les types des paramètres, $m(t)$ devenant $m_t(t)$ et $m(u)$, $m_u(u)$ — pour faire disparaître la surcharge statique sans rien changer au comportement d'ensemble. Ce nouveau nommage étant non ambigu (injectif), il peut alors servir à identifier les entités génériques. Le renommage d'un appel de méthode se fait dans l'espace de noms du type statique du receveur, en tenant compte des types statiques des paramètres.

Au total, on a donc une opération en 3 temps, avec les sélections successives : 1) de l'ensemble des entités génériques de nom m du type *statique* τ_s du receveur ; 2) dans cet ensemble, de l'entité générique m_t adaptée au type *statique* des arguments et introduite dans la classe A ; 3) à l'exécution et dans l'entité générique m_t, de la méthode m_t_B correspondant au type *dynamique* τ_d du receveur. La sélection met donc en jeu 4 types différents, partiellement ordonnés par $\tau_d <: B <: A$ et $\tau_d <: \tau_s <: A$. Entre le type dynamique du receveur et le type qui introduit l'entité générique, on trouve, *a priori* incomparables par la relation de spécialisation, la classe qui possède la méthode héritée et le type statique du receveur (figure 1).

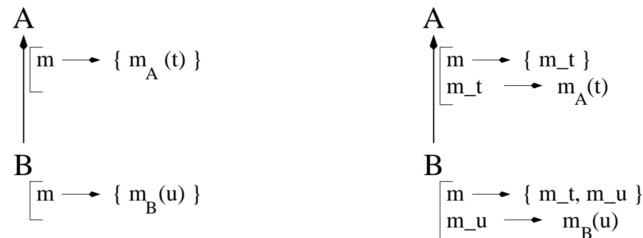

Figure 2 – Surcharge statique. En C++ (à gauche), chaque nom est lié à un ensemble de méthodes surchargées et la spécialisation provoque un masquage des noms. En JAVA (à droite), chaque nom est lié à un ensemble d'entités génériques que la spécialisation cumule, le masquage s'effectuant au niveau de chaque entité générique.

3.1.3. *Cas de* C++

Le modèle précédent, qui découle d'une analyse abstraite, est aussi celui de JAVA. Le modèle de C++ en diffère assez pour qu'il soit nécessaire de le décrire. Soit une classe A avec une méthode $m_A(t)$, et une sous-classe B de A définissant une méthode $m_B(u)$ de même nom m, mais de type de paramètre $u \neq t$: en C++, la méthode $m_A(t)$ n'est plus accessible sur les objets de type statique B, à moins que $m_B(t)$ ne soit aussi redéfinie. La sémantique de la surcharge statique ne respecte donc pas l'invariance par renommage que nous avons avancée : le renommage de m_B en m_u_B ferait en effet réapparaître m_t_A dans la portée de la classe B. La sélection s'effectue pourtant bien suivant les 3 étapes décrites plus haut mais la surcharge empêche l'héritage des méthodes de même nom. Ainsi, en C++, l'ensemble des entités attachées à un nom est hérité, sauf si ce nom est redéfini dans la sous-classe : le phénomène de *masquage* qui caractérise la redéfinition n'est pas attaché aux entités génériques (m_t) mais aux noms (m) (cf. figure 2). Autre formulation : l'intension serait constituée de noms, pas d'entités génériques. Ce qui revient en fait à nier l'existence de ces entités génériques. En C++, une classe n'est guère plus qu'un espace de noms : *Note that dominance* [i. e. masquage] *applies to names not just to functions* [ELL 90, p. 205].

3.2. *La sélection multiple : le cas* CLOS

La sélection multiple a été introduite par COMMON LOOPS [STE 86], puis popularisée par CLOS [BOB 88] avant d'être introduite dans un cadre de typage statique par [MUG 91] et théorisée par [GHE 91, CAS 96b, CAS 97]. Elle a été reprise par différents langages comme CECIL [CHA 93], DYLAN [App95], ou CLAIRE [CAS 99]. Dans la sélection multiple, la sélection dynamique de la méthode à appliquer, c'est-à-dire la *liaison tardive*, ne se fait pas sur un seul argument — le receveur du message, `self`, `current` ou `this` suivant les langages — mais sur tous les paramètres de la méthode. La sélection multiple ressemble ainsi à la surcharge, mais c'est une surcharge exclusivement dynamique : toutes les méthodes de même nom appartiennent

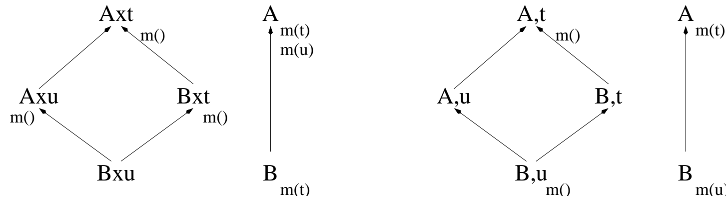

Figure 3 – Les deux visions de la sélection multiple : il manque, à gauche, pour le produit, $m_{B\times u}$ et, à droite, pour le sous-typage des multi-méthodes, $m_{B,t}$.

à la même entité générique qui prend, en CLOS, le nom de *fonction générique* (cf. section 3.1). La règle de la contravariance, qui ne s'applique qu'aux paramètres secondaires des méthodes, disparaît en même temps que ces paramètres secondaires. La sélection multiple se ramène à la sélection simple de deux façons différentes.

3.2.1. *Sélection simple dans le produit des types*

Dans la première, la méthode n'est plus encapsulée dans une classe mais leur est transversale. Tout se passe comme si la méthode n'avait qu'un seul paramètre — le receveur du message — dans le produit des types [DUC 89]. On remplace la notation $m_A(t)$ par $m_{A\times t}$ qui montre bien l'absence de distinction entre le receveur et les autres arguments. Le sous-typage du type produit s'exprime ainsi :

$$u_1 \times u_2 <: t_1 \times t_2 \iff u_1 <: t_1 \ \& \ u_2 <: t_2 \qquad (8)$$

C'est la vision implicite de CLOS. Le problème de la covariance disparaît mais il reste un problème résiduel qui relève du conflit d'héritage multiple. En effet, si $m_{B\times t}$ et $m_{A\times u}$ sont définies, quelle sera la méthode sélectionnée pour un receveur du type dynamique $B \times u$ (figure 3, gauche) ? La règle de typage de [CAS 96b], qui impose l'existence d'une borne inférieure, suffit pour résoudre ce problème : cette borne inférieure sera le résultat de la sélection.

3.2.2. *Multi-méthodes et sélection en deux temps*

Il est aussi possible de conserver la modularité liée à la notion de receveur et à l'encapsulation des méthodes dans les classes : on associe à chaque classe une *multi-méthode*[2], regroupant toutes les méthodes de même nom ayant la classe comme type du premier paramètre et variant par le type des autres paramètres. La notation $m_{A,t}$ est alors plus appropriée pour souligner l'ordre entre le receveur et les autres arguments. La sélection s'effectue en deux temps, un peu comme pour la surcharge statique (section 3.1.2) mais de façon purement dynamique : 1) sélection de la multi-méthode suivant le type dynamique du receveur ; 2) sélection de la méthode dans la multi-méthode suivant le type dynamique des paramètres secondaires, dans le produit de leurs types.

2. Cet usage du terme est différent de celui qui est pratiqué par COMMON LOOPS ou CLOS, où la multi-méthode est la méthode dont la sélection est basée sur tous les paramètres.

	t	$t[u]$	u		
A	$m_A(t)$	$m_A(t)$	$m_A(t)$	surcharge	
$A[B]$	$m_A(t)$	$m_A(t)$	$m_A(t)$	statique	C++
B	$+-$	$+-$	$m_B(u)$	incorrecte	
A	$m_A(t)$	$m_A(t)$	$m_A(t)$	surcharge	
$A[B]$	$m_A(t)$	$m_A(t)$	$m_A(t)$	statique	JAVA
B	$m_A(t)$	$m_A(t)$	$m_B(u)$	correcte	
A	$m_{A\times t}$	$m_{A\times t}$	$m_{A\times t}$	sélection	
$A[B]$	$m_{A\times t}$	$m_{B\times u}$	$m_{B\times u}$	multiple	CLOS
B	$m_{A\times t}$	$m_{B\times u}$	$m_{B\times u}$		
A	$m_A(t)$	$m_A(t)$	$m_A(t)$	covariance	
$A[B]$	$+*$	$m_B(u)*$	$m_B(u)$	avec typage	EIFFEL
B	$+-$	$-$	$m_B(u)$	statique	
A	$m_A(t)$	$m_A(t)$	$m_A(t)$	covariance	
$A[B]$	$+$	$m_B(u)$	$m_B(u)$	avec typage	
B	$+$	$m_B(u)$	$m_B(u)$	dynamique	

Figure 4 – La sélection des méthodes $m_A(t)$ et $m_B(u)$ suivant les types statiques et dynamiques (ces derniers entre crochets, lorsqu'ils diffèrent du type statique) du receveur (lignes) et du paramètre (colonnes), dans un contexte où $B \prec A$ et $u <: t$.

Cette vision de la sélection multiple est celle de [MUG 91, CAS 96b]. Le typage sûr est assuré par des règles complémentaires pour les multi-méthodes : si $m_{A,t}$ est définie et que m est redéfinie en B, il faut qu'il existe t' tel que $t <: t'$ et $m_{B,t'}$ soit définie. Dans l'exemple de la figure 3, à droite, il faut définir $m_{B,t}$.

Modulo des prises en compte orthogonales de l'encapsulation, ces deux visions de la sélection multiple coïncident, dès lors qu'elles sont munies des règles qui leur assurent un typage sûr : s'il existe une borne inférieure (produit des types), elle sera sélectionnée (multi-méthodes). En sens inverse, les règles de sous-typage des multi-méthodes imposent de rajouter des méthodes qui peuvent être héritées dans le produit mais $m_{B,t}$ peut se contenter de faire appel à $m_{A,t}$.

3.3. *Synthèse sur la sélection*

La figure 4 présente les différents cas de sélection, avec un exemple typique de deux méthodes $m_A(t)$ et $m_B(u)$, où B est une sous-classe de A et u un sous-type de t. Chaque cas est décrit par un tableau faisant varier les types statiques et dynamiques du receveur (lignes) et de l'argument (colonnes) et indiquant la méthode sélectionnée ou l'erreur de type résultant. Le "$+$" (resp. "$-$") désigne une erreur de type détectée dynamiquement (resp. statiquement) : "$+-$" désigne donc une erreur de type dynamique détectée statiquement, "$-$" seul une erreur statique qui n'aurait pas causée d'erreur dynamique et "$+$" seul une erreur détectée par une vérification dynamique. Le tableau est complété par les deux traitements de la covariance : en typage dynamique, c'est-à-dire sans vérifications statiques, et en typage statique, comme en EIFFEL. Dans ce

	t	$t[u]$	u		
A	$m_A(t)$	$m_A(t)$	$m_A(t)$	surcharge	C++
$A[B]$	$m_B(t)$	$m_B(t)$	$m_B(t)$	statique	JAVA
B	$m_B(t)$	$m_B(t)$	$m_B(u)$		
A	$m_{A\times t}$	$m_{A\times t}$	$m_{A\times t}$	sélection	
$A[B]$	$m_{B\times t}$	$m_{B\times u}$	$m_{B\times u}$	multiple	CLOS
B	$m_{B\times t}$	$m_{B\times u}$	$m_{B\times u}$		

Figure 5 – Surcharge statique et sélection multiple : cas de 3 méthodes.

dernier cas, le "$*$" désigne les *catcalls* polymorphes : s'ils sont interdits, c'est une erreur à la compilation ("$-$"), mais une inférence de types plus sophistiquée permettrait d'en conserver certains (cf. section 4.4). La figure 5 présente le même exemple, avec une troisième méthode $m_B(t)$, dans les 2 cas qui l'autorisent : les surcharges statiques de C++ et JAVA — qui, dans ce cas, coïncident — et la sélection multiple.

4. Simulation de la covariance et du typage sûr

Aucun langage standard n'ose enfreindre le dogme du typage sûr : qu'ils simulent la covariance (C++ ou JAVA) ou l'appliquent (EIFFEL), ils prétendent rester sûrs.

4.1. *La coercition de types*

Quasiment tous les langages de programmation qui prétendent leur typage sûr disposent en fait d'un moyen imparable pour déclencher des erreurs de type : la coercition (*casting*), que nous considérons uniquement dans sa version descendante, ou *downcast*. En C++, il s'agit de la coercition effectuée par `dynamic_cast` dans le cas où le type cible est un sous-type strict du type statique de l'argument.

4.1.1. *Coercition des paramètres covariants*

Soit la méthode $m_A(t)$ de la classe A, redéfinie dans sa sous-classe B par une méthode $m_B(t)$, de façon invariante alors que la spécialisation sous-jacente est strictement covariante et nécessiterait une méthode $m_B(u)$ avec $u <: t$: le type statique du paramètre de m_B est t, alors que son type dynamique, suivant la spécification, est, ou doit être, un sous-type de u. La spécification peut donc imposer d'appliquer à ce paramètre une opération connue par le type u mais pas par le type t, ce que le typage statique usuel interdit. Il faut donc indiquer au compilateur, par un *cast*, que ce t doit être un u et peut être traité comme tel. C'est une hypothèse non prouvée qui nécessitera une vérification à l'exécution, qui déclenchera, éventuellement, une erreur de type. Ainsi, le malheureux `Phoque`, dont la méthode `manger` hérite de `Animal` un paramètre d'un type invariant `N'importe-quoi`, ne peut `manger` ce `N'importe-quoi` qu'en le « prenant pour » un `Poisson`, par exemple en le faisant

`nager` avant pour vérifier qu'il est bien frais. Or la méthode `nager` ne fait pas partie de l'interface de `N'importe-quoi` et sera donc refusée par le compilateur si on ne lui dit pas de prendre ce `N'importe-quoi` pour un `Poisson`.

Le *casting* peut prendre des formes diverses. Dans certains langages, il s'exprimera par la construction `typecase` (SATHER) ou par des *tentatives d'affectation* (EIFFEL). Il peut donner lieu à des erreurs de type effectives (signalement d'exception) à l'exécution (JAVA, `dynamic_cast` de C++), à des erreurs indéterminées sans signalement d'exception (`static_cast` de C++) ou les dissimuler sous le voile pudique de la valeur `void`[3] (EIFFEL). Mais, dans tous les cas, ce *casting* constitue de fait un typage dynamique, borné par le type statique de l'entité considérée. À l'exception d'EIFFEL, il sert essentiellement à compenser l'interdiction de la covariance. Bien que ce soit argumenté avec méthode par les différents auteurs, c'est une construction générale, utilisable à tort et à travers par le programmeur : s'il y a un bon usage du *casting*, ce bon usage reste, par définition, « subjectif ». Par ailleurs, lorsque la spécification relève d'une spécialisation covariante, sa mise en œuvre dans un langage contravariant par le biais du *casting* fait perdre au typage le rôle de commentaire revendiqué par [MEY 97] ou, pire, en fait un commentaire inadéquat.

Ainsi, à vouloir faire sortir les erreurs de type par la grande porte, on n'aboutit qu'à les faire rentrer par la porte de service, avec la circonstance aggravante que ce trait de langage n'étant pas très avouable et considéré comme marginal, ses performances n'ont pas forcément été très optimisées. Or la vérification dynamique de type peut coûter très cher [HAB 97, VIT 97, RAY 01].

REMARQUE. — L'existence du *casting* descendant se justifie aussi par l'impossibilité de rajouter de nouvelles méthodes aux classes préexistantes : lorsque l'on veut effectuer des traitements différents suivant le type dynamique d'un objet, seule une construction comme `typecase` permet de distribuer le traitement sur les différents types comme l'aurait fait une méthode *ad hoc*. Le cas est particulièrement fréquent avec les fonctions de lecture qui peuvent lire et retourner un objet de n'importe quel type. Cette limitation du modèle objet standard n'existe pas dans les langages disposant de la sélection multiple : un `typecase` sur les types A_1,..,A_n, dans la classe C, peut être remplacé par une multi-méthode avec les méthodes m_{C,A_1},..,m_{C,A_n}.

4.1.2. *Coercition des attributs covariants*

Le *casting* constitue donc un moyen d'implémenter une spécialisation covariante des paramètres de méthodes, en réintroduisant le risque d'erreurs de types. Pour les attributs, il est possible de faire un *cast* à chaque accès en écriture mais la définition d'*accesseurs* permettra de rendre cette simulation moins subjective. L'accesseur en écriture sera redéfini (avec un type de paramètre invariant) à chaque redéfinition du type de l'attribut en lui faisant faire le *cast* et la vérification de type nécessaire, comme pour les paramètres de méthodes covariants. Mais il serait préférable d'éviter ces re-

3. `Void` constitue un risque perpétuel d'erreur de type : les langages considèrent que `void` est de tous les types car c'est la seule façon de pouvoir affirmer qu'ils sont sûrs. Mais c'est absurde, car le sous-type de tous les types est un type absurde, $\perp$, qui n'a pas d'instance.

définitions en factorisant ces vérifications au moyen d'un attribut « de classe » qui définirait le type de l'attribut dans la classe. A défaut d'un attribut de classe, qu'il ne faut pas confondre avec les variables `static` de ces langages, ni même avec ceux de SMALLTALK, une méthode retournant ce type fera parfaitement l'affaire. Mais encore faut-il que les types soient des valeurs de première classe et que le langage considéré permette de les comparer avec le type dynamique d'une valeur.

En théorie, un unique accesseur en écriture devrait suffire. En pratique ce n'est pas le cas, car l'implémentation peut nécessiter de faire un ajustement de pointeur suivant le type statique des références [ELL 90, DUC 01a]. De plus, pour que cette simulation soit la plus « objective » possible, c'est-à-dire qu'elle ne dépende pas de la bonne volonté du programmeur en autorisant un mauvais usage de l'attribut, l'attribut ne devrait plus être manipulable autrement que par ses accesseurs. Pour ces raisons, un accesseur en lecture sera défini, qui fera le *cast* nécessaire, sans vérification de type. Il aura un type de retour covariant car cela donne une information de type correcte au compilateur : l'absence de redéfinition entraînerait une perte d'information. Il faudrait aussi que l'attribut soit inaccessible dans les sous-classes, ce qui peut être obtenu par le mot-clé `private` mais impose l'accesseur en lecture. De plus, les redéfinitions des accesseurs ne pouvant utiliser l'attribut devraient faire appel explicitement à l'accesseur de la classe de définition de l'attribut, ce qui est possible en C++ mais ne l'est pas en JAVA qui imposerait une cascade de `super`.

Bien entendu, ces accesseurs doivent être définis à la main, à moins d'utiliser un préprocesseur ou méta-niveau comme OPENC++ ou OPENJAVA qui peut ajouter le sucre syntaxique nécessaire au confort du programmeur [CHI 95]. L'usage de la méta-programmation est d'ailleurs la seule façon d'éviter des redondances, ne serait-ce qu'entre le type en lecture et le type en écriture. En son absence, la simulation est possible, mais terriblement besogneuse.

4.2. *Simulation par la surcharge statique*

Si la surcharge statique permet des définitions de méthodes avec n'importe quels types de paramètres et de valeur de retour, elle permet aussi des définitions dans les sous-classes, qui ressemblent terriblement à des *redéfinitions covariantes*. Mais c'est un leurre. Soit l'exemple de la figure 4 : on appellera $m_B(u)$ sur un receveur de type statique B avec un argument de type statique u et $m_A(t)$ dans tous les autres cas, *même si le receveur est une instance de* B *et le paramètre une instance de* u. A notre connaissance, l'usage de la surcharge statique pour simuler la covariance n'a pas été défendu explicitement, mais on trouve des traces d'une pratique *ad hoc* [PéT 97]. Pourtant, un compilateur ne peut pas assurer que $m(u)$ sera appliquée à toute instance de B : s'il le pouvait, il pourrait aussi bien accepter une redéfinition covariante. Pour que la simulation soit complète, il faut encore définir $m_B(t)$ en lui faisant appeler $m_B(u)$ avec un *cast* sur son paramètre.

Une spécificité de JAVA augmente encore la confusion. La surcharge statique provoque en effet des interférences avec la sémantique de `super`, qui permet à une mé-

thode de faire appel à la méthode (en première approximation, de même nom) définie dans ses super-classes. Il s'agit donc d'un analogue du `super` de SMALLTALK, du `call-next-method` de CLOS, ou du `precursor` d'EIFFEL. Comme en SMALLTALK et contrairement à CLOS ou EIFFEL, le nom de la méthode figure dans l'appel à `super`. Dans l'exemple de la figure 4, un appel à `super`.m sur son propre paramètre (de type statique u) par la méthode $m_B(u)$ provoque l'appel de la méthode $m_A(t)$. La simulation n'en est que plus parfaite, `super` donnant l'impression que $m_A(t)$ et $m_B(u)$ appartiennent à la même entité générique. Le problème est ici de savoir comment doit se comprendre la référence à m de `super`.m dans le corps de la méthode m_u : faut-il le renommer en m_u ou m_t ? Il serait légitime que ce soit en `super`.m_u, la spécification naturelle et l'usage le plus courant consistant à n'utiliser `super` que dans le cadre de l'entité générique courante, donc sur le nom de la méthode courante, en un équivalent assez exact de `call-next-method` ou de `precursor`. Et `super`.m_u échouerait puisque m_u_A n'est pas défini. Or cette interprétation naturelle n'est pas celle de JAVA où `super` peut s'appliquer à un nom de méthode différent de celui de la méthode courante : et l'on voit bien, ici, qu'il s'agit de nom et non pas des entités génériques qu'elles désignent. Ce cas est choquant parce qu'il s'agit de `super` mais ce comportement s'explique simplement si l'on considère que `super` a pour type statique implicite, non pas la classe qui définit la méthode, comme `self`, mais sa super-classe : A et non pas B. Il est alors légitime de procéder à la désambiguïsation dans le contexte du type statique du receveur (étape 2 de la sélection). Il faut critiquer ici le choix de SMALLTALK et JAVA de faire de `super` une pseudo-variable, comme `self`, au lieu d'en faire un mécanisme fonctionnel bien identifié comme `call-next-method` ou `precursor`. En effet, si `self` est conceptuellement bien défini comme l'objet receveur du message, `super` n'est qu'un point de vue un peu confus sur ce receveur : l'une et l'autre variables désignent en fait le même objet.

4.3. *Simulation par la sélection multiple*

La sélection multiple semble résoudre le problème de la covariance pour les méthodes. [BOY 96] propose ainsi une transformation automatique d'une sélection simple avec redéfinition covariante à une sélection multiple. Elle consiste à considérer chaque méthode comme une multi-méthode et à rajouter les branches nécessaires pour éviter les erreurs de types. Dans notre exemple récurrent, la technique transforme les 2 méthodes $m_A(t)$ et $m_B(u)$, en $m_{A,t}$ et $m_{B,u}$ et rajoute une branche $m_{B,t}$ à la multi-méthode de B. Pour [BOY 96], cette nouvelle branche $m_{B,t}$ est une copie de ou un appel à $m_{A,t}$. Mais cette transformation n'est pas sans effet sur le comportement du programme, même si le seul changement concerne les cas où une erreur de type se serait produite à l'exécution. Pour revenir à notre `Phoque`, ne changer que le mode de sélection lui permettrait de manger `N'importe-quoi` avec la méthode $\texttt{manger}_{\texttt{Animal,N'importe-quoi}}$, ce qui n'est pas l'objectif. Pour obtenir le comportement désiré, il faudrait rajouter une méthode $\texttt{manger}_{\texttt{Phoque,N'importe-quoi}}$ qui signalerait une exception. On pourrait appeler cela une « erreur de type », par exemple !

La sélection multiple offre donc une solution à la covariance pour les méthodes. Cette solution subjective, le programmeur devant définir lui-même les branches qui causent une erreur de type, peut être rendue objective par la même technique que [BOY 96], mais en faisant signaler des exceptions à toutes les branches rajoutées. Si le langage d'origine est à sélection simple, cela ne serait qu'une implémentation, pas forcément la plus efficace, d'un langage covariant au moyen de la sélection multiple. Comme il serait dommage d'utiliser la sélection multiple sans qu'elle participe à l'expressivité du langage, il faudrait pouvoir exprimer à la fois la covariance et la sélection multiple. La technique de [BOY 96] s'impose, malgré la confusion qu'elle peut apporter dans l'esprit des programmeurs : les règles de typage de [CAS 97] ne sont plus imposées au programmeur mais le compilateur les utilise pour générer les branches manquantes, qui signalent une erreur de type. Cependant, malgré les tentatives de [CAS 96a], la sélection multiple ne s'applique pas aux attributs.

4.4. *La covariance sûre : le cas d'*EIFFEL

EIFFEL est le seul langage de programmation par objets fortement typé à refuser sciemment la règle de contravariance, qui est remplacée, dans tous les cas — attribut ou paramètre de méthode — par une règle de covariance qui en est l'inverse exact. Les mécanismes d'encapsulation (clauses d'export) du langage permettent même une violation de l'inclusion des intensions (2) mais nous ne considérerons pas ce dernier point ici. EIFFEL privilégie donc le pouvoir expressif requis par la spécialisation, avec une argumentation très empirique : *Covariance is, according to all available evidence, what we need in practice* [MEY 97, p 624].

Un langage covariant peut donner lieu à des erreurs de type dans deux cas appelés *catcalls* — où « cat » est l'acronyme de *Changing Availability or Type* —, dont seule la redéfinition covariante nous intéresse ici. Pour conserver un typage sûr, [MEY 97] propose la règle suivante : 1) un *catcall* est un appel d'une propriété (attribut ou méthode) dont l'un des paramètres est redéfini de façon covariante ; 2) un appel est polymorphe lorsque la propriété est appelée sur une expression d'un certain type dont la valeur peut être d'un sous-type strict ; 3) les *catcalls* polymorphes sont interdits. Si une classe A a une méthode $m(t)$ et qu'une sous-classe B redéfinit $m(u)$ avec $u <: t$, alors l'appel $x.m(arg)$ est interdit si les types statiques de x et arg sont respectivement A et t, et s'il n'est pas prouvé que x est de type dynamique A. On ne peut pas utiliser les méthodes redéfinies de façon covariantes sur des receveurs potentiellement polymorphes, charge au compilateur de prouver qu'un argument ne peut pas être polymorphe. Cette règle évite bien les erreurs de type mais elle est tellement pessimiste qu'elle réduit grandement l'intérêt de la spécialisation covariante. En effet, à part les types non spécialisés, une seule chose échappe au polymorphisme dans ces définitions précédentes : les variables locales dont la valeur provient directement d'une instanciation de leur type statique. Tout paramètre formel et tout appel de fonction est potentiellement polymorphe. Autant dire que, pour une méthode redéfinie de façon covariante, à peu près tous les usages sont interdits.

4.5. *Conclusion provisoire*

Malgré leur orthodoxie de typage, tous ces langages permettent une certaine pratique de la redéfinition covariante avec les risques d'erreur de types qui lui sont inhérents. En C++ ou JAVA, le *casting* est un opérateur général qui peut faire beaucoup plus de dégats qu'une redéfinition covariante bien conçue. De plus, la surcharge statique apporte un élément de confusion considérable dans l'esprit des programmeurs (voir aussi l'analyse de [BEU 02]). Des défauts de ces langages empêchent une simulation propre de la covariance, mais la correction de ces défauts ne suffirait pas à rendre convaincante une simulation qui resterait trop subjective. La sélection multiple, de son côté, offre une alternative satisfaisante, à condition de détourner les règles de sous-typage de [CAS 97] et pour les méthodes seulement. Mais CLOS est le seul langage d'usage relativement courant à mettre en œuvre la sélection multiple, dans un cadre impropre pour nos objectifs puisqu'il s'agit de typage dynamique et d'un modèle de fonctions génériques orthogonales aux classes. Si l'objectif est la covariance, autant définir des langages covariants en sélection simple, dont l'implémentation sera plus efficace. Quant à EIFFEL, il offre le système de types le plus adéquat — sans doute perfectible — mais il maintient, au moins en théorie, l'illusion d'un typage sûr.

5. Variations autour du polymorphisme et de la spécialisation

Jusqu'ici, nous nous sommes basés sur un système de types implicite relativement simple, avec une identification aux classes. Dans cette section, nous examinons diverses variations : autour du sous-typage avec d'autres relations de conformité comme le filtrage (*matching*), autour du polymorphisme, avec une comparaison entre les polymorphismes paramétrique et d'inclusion, autour de la covariance avec les types ancrés, et enfin, autour de la spécialisation en la dissociant du sous-typage.

5.1. *Variations autour du sous-typage*

Nous avons défini le sous-typage comme une relation de conformité entre types, un préordre, qui assure la substituabilité du polymorphisme d'inclusion. D'autres préordres sont envisageables si l'on abandonne cette propriété.

5.1.1. *Les types récursifs et `mytype`*

La plupart des classes apparaissent récursivement dans leur description, dès qu'elles servent à typer l'une de leurs propriétés, ce qui impose de modéliser leur type par un type récursif, de la forme $\mu(t)A(t)$, où μ est opérateur de point fixe et le paramètre t désigne le type en cours de définition : $\mu(t)A(t) = A(\mu(t)A(t))$. Dans les langages de programmation, ce paramètre formel est désigné par `mytype` (en OBJECTIVE-CAML ou POLYTOIL [BRU 95]), `same` (SATHER) ou `like current` (EIFFEL) : comme `self`, `mytype` n'a de sens que dans la portée, symbolisée par le constructeur μ, de

la définition de la classe courante. Le sous-typage (sûr) d'un type récursif est défini à partir du sous-typage des types non récursifs, par la règle [AMA 93, ABA 95] :

$$\mu(t)B(t) <: \mu(t)A(t) \text{ ssi } u <: t \Rightarrow B(u) <: A(t). \tag{9}$$

Il faut donc que le paramètre t n'apparaisse qu'en position covariante dans $A(t)$. Utilisé ainsi, il résout le problème connu sous le nom de *perte d'information*, pour les méthodes qui retournent `self` ou en font une copie. En position contravariante (type de paramètre), `mytype` offre bien une réponse au problème des *méthodes binaires*, dont le paramètre est de même type que le receveur, typiquement les méthodes d'égalité, mais cette réponse n'est pas sûre.

5.1.2. *Les types ancrés et la covariance*

[MEY 97] remarque que `current` — c'est-à-dire l'objet récepteur du message en EIFFEL, le `self` de SMALLTALK ou SATHER, le `this` de C++ ou JAVA — est typiquement covariant, donc `mytype` aussi. La redéfinition covariante conduit à des constructions de types basées sur le type du receveur et utilisées aussi bien en position covariante que contravariante. Ces constructions sont appelées *types ancrés* en EIFFEL. Outre `like current`, c'est-à-dire `mytype`, EIFFEL étend cette notion en autorisant comme ancre n'importe quelle propriété de `current` (`like` p). Rien ne semble empêcher de généraliser les ancres à un chemin de propriétés : `like` $p_1 ... p_n$. Ces types ancrés sont le seul moyen de redéfinir le type d'une méthode sans avoir à redéfinir son code, donc en assurant un parfait héritage d'implémentation.

De la même manière que `mytype`, `like` p n'est sûr que s'il n'est utilisé qu'en type de retour et si p est une méthode. Lorsque p est un attribut, le type ancré repose lui-même sur une redéfinition non sûre. Pourtant, les types ancrés sont sûrs tant qu'ils ne sortent pas de la portée de l'opérateur μ et qu'ils restent ancrés sur `self` : pour un objet extérieur, *a fortiori* pour une classe, l'ancre ne signifie rien et ne peut être rapportée qu'au type *statique* du receveur, d'où le risque d'erreur pour un paramètre ou un attribut en écriture. Plus généralement, lorsqu'elles sont ancrées sur le même receveur, deux occurrences d'un même type ancré sont comparables : il est alors son seul sous-type sûr. Ainsi, le *catcall* polymorphe $x.m(x.n(arg))$ serait sûr si m et n étaient typés respectivement par $m($`like` $p)$ et $n()$: `like` p pour quelque p, alors qu'il ne le serait pas avec des types $m(B)$ et $n() : B$, dès lors que m aurait des redéfinitions covariantes. Sa bonne utilisation peut donc permettre de réduire, voire éliminer, les erreurs de type à l'intérieur d'un type, en les cantonnant à la frontière entre types. Typiquement, dans la définition d'une classe A, on typera par `like current` au lieu de A et, si cette classe a un attribut p de type B, on typera les autres propriétés par `like` p au lieu de B. Les types ancrés contribuent donc à réduire les risques d'erreur de types entraînés par la covariance, sans les annuler néanmoins.

5.1.3. *Le filtrage (*matching*) et le sous-typage covariant*

Le filtrage ou *matching* [BRU 95, ABA 95, PLA 98] est une relation entre types récursifs, notée <#, plus faible que le sous-typage (9) :

$$\mu(t)B(t) <\# \mu(t)A(t) \text{ ssi } B(t) <: A(t). \tag{10}$$

Le filtrage autorise l'usage de `mytype` en position contravariante et pas simplement en position covariante, et c'est sa seule infraction à la règle de contravariance. Sans cet usage de `mytype`, le filtrage s'identifie au sous-typage.

La relation de sous-typage $<:$ sur les types *record* non récursifs est définie à partir de la règle de contravariance : B est un sous-type de A si chaque champ de A appartient aussi à B, avec le même type s'il s'agit d'un attribut et un sous-type fonctionnel s'il s'agit d'une méthode. En remplaçant la règle de contravariance par la règle de covariance induite par l'inclusion des domaines (3), on obtient une relation $<!$ que l'on peut étendre aux types récursifs de la même manière que (9).

$$\mu(t)B(t) <! \, \mu(t)A(t) \text{ ssi } u <! \, t \Rightarrow B(u) <! \, A(t). \qquad (11)$$

Un analogue du *matching*, basé sur $<!$ à la place de $<:$ serait identique à la relation précédente car $B(t) <! \, A(t)$ ssi $u <! \, t \Rightarrow B(u) <! \, A(t)$. On dispose ainsi de 3 relations entre types : le sous-typage, qui est la plus grande relation sûre, le *matching* et le sous-typage covariant. Si l'on restreint la règle de contravariance à l'invariance du type des paramètres, ces trois relations sont incluses les unes dans les autres : $<: \; \subseteq <\# \subseteq <!$.

5.2. *Types paramétrés*

Le polymorphisme paramétrique (ou généricité) repose lui aussi sur une notion de substituabilité mais elle diffère sensiblement du sous-typage.

5.2.1. *Généricité bornée*

La généricité bornée repose sur l'expression de types paramétrés $A\langle T <: B\rangle$ dont le paramètre T n'est instanciable que par un sous-type d'une borne B : la théorie considère alors que le type formel T est dans la portée d'une quantification universelle, exprimant ainsi une certaine notion de substituabilité. La généricité *F-bornée* (*F-bounded polymorphism*) consiste en ce que la borne soit elle-même paramétrée par le même type formel [CAN 89]. L'exemple classique est celui des ensembles ordonnés qui s'exprime par $\texttt{orderedSet}\langle T <: \texttt{Comparable}\langle T\rangle\rangle$. Le *matching* permet une expression équivalente, $\texttt{orderedSet}\langle T <\# \, \texttt{Comparable}\rangle$: les occurrences de T dans $\texttt{Comparable}\langle T\rangle$ sont alors remplacées par `mytype`.

En effet, si le type $A\langle B\rangle$ n'introduit pas d'erreurs de type, il en est de même pour $A\langle C\rangle$ si $C <: B$, mais aussi si $C <\# \, B$. On pourra dire que ces relations $<:$ et $<\#$ supportent une *généricité bornée sûre*. En corollaire, on constate que la substituabilité associée au polymorphisme paramétrique repose sur une relation de conformité plus large que celle qui est associée au polymorphisme d'inclusion. En élargissant encore la relation entre types jusqu'à $<!$, on perd la sûreté de la généricité bornée : si $C <! \, B$, c'est-à-dire si la classe C redéfinit de façon covariante, autrement que par `mytype`, des propriétés de B, l'utilisation de ces propriétés dans A ne sera plus sûre. Dans un langage covariant, la relation de sous-typage entre classes, $<!$, est utilisée pour borner la généricité. La définition du type paramétré $A\langle T <! \, B\rangle$ produira donc des erreurs sur certaines instanciations. Une partie de ces erreurs pourrait être évitée, si A pouvait se

servir d'un type `like` $p.T$, ancré sur le type formel T, pour les propriétés p de B qui sont redéfinies de façon covariante.

Enfin, la relation de conformité entre instances différentes d'un même type paramétré ne tient pas toujours. Si $C <: B$, la relation $A\langle C\rangle <: A\langle B\rangle$ est en général fausse, de même que $A\langle C\rangle <\# A\langle B\rangle$. En revanche, la relation $<!$ a l'avantage de vérifier $C <! B \Rightarrow A\langle C\rangle <! A\langle B\rangle$. Il est donc cohérent qu'EIFFEL accepte cette relation de sous-typage entre instances de types paramétrés, même si la substitution d'un $A\langle C\rangle$ à un $A\langle B\rangle$ n'est pas plus sûre que la substitution d'un C à un B.

5.2.2. *Types paramétrés, types ancrés et types virtuels*

Les types récursifs ont montré qu'un système de types suffisamment expressif doit être formalisé par des types paramétrés. Le cas du type ancré `like` p peut aussi s'exprimer par un type paramétré, en typant par un paramètre formel T aussi bien p que les entités typées `like` p. On y gagne une symétrie bien mal exprimée par le type ancré : comparez $\nu(t)\{p : t;\ q : t\}$ avec $\{p : t;\ q : \texttt{like}\ p\}$. Les corrections du système de types d'EIFFEL proposées par [COO 89, WEB 92] reposent sur cette proximité entre les types ancrés et les types paramétrés. L'objectif de ces auteurs est de ramener EIFFEL dans le droit chemin du typage sûr : ce n'est pas le nôtre. La proposition de [PAL 90], basée sur la substitution de types, pousse à l'extrême cette approche consistant à interpréter un type comme un type paramétré : tout type figurant dans une expression de type peut potentiellement servir de paramètre.

Inversement, toute classe paramétrée bornée $A\langle T <! B\rangle$ peut s'exprimer par une classe non paramétrée A_B, en remplaçant chaque occurrence du type formel T par un type ancré `like` p sur un attribut p de type B. L'instanciation $A\langle B\rangle$ est identique à A_B et $A\langle C\rangle$ est obtenue en spécialisant A_B par A_C où p est redéfini de type C. La relation $A_C <! A_B$ est aussi sûre, ou peu sûre, que $A\langle C\rangle <! A\langle B\rangle$. Les types ancrés constituent ainsi une alternative aux types paramétrés, assez similaire aux *types virtuels* proposés dans le langage BETA et discutés dans [BRU 98]. La différence, qui semble la seule justification de la présence de la généricité en EIFFEL, est que l'instanciation d'un type paramétré est automatique, alors que la spécialisation d'une classe EIFFEL ou d'un type virtuel est forcément explicite.

5.3. *Distinguer spécialisation et sous-typage*

5.3.1. *Distinguer classes et types*

Lorsqu'un langage distingue classes et types, les types sont définis par un ensemble de signatures et le type d'une classe est généralement constitué par le sous-ensemble des signatures de sa partie publique. Etant donné une règle de conformité, il est possible de *calculer* la relation de sous-typage au lieu d'imposer de l'expliciter. Ainsi, les clauses `where` de CLU ou THETA [DAY 95] constituent un typage avec sous-typage implicite, dans un usage restreint à la généricité. Si la relation de sous-typage n'est pas calculée, il est possible de s'en approcher en permettant de définir *a posteriori* des super-types de types ou classes existants, autrement que par spécialisation, ce qui

augmente nettement la réutilisabilité et la factorisation. C'est le cas de SATHER avec un constructeur de super-types et de CLAIRE, qui possède un opérateur de disjonction sur les types. *A contrario*, les interfaces de JAVA ne peuvent être définies après les classes qu'elles implémentent, ce qui constitue une limitation certaine. [HUC 00] montre pourtant comment calculer automatiquement les interfaces JAVA qui typent implicitement un ensemble de classes. Sur ce point, le seul apport de JAVA est de l'ordre de l'héritage multiple : la relation d'héritage est multiple quand on considère les interfaces, mais, restreinte aux classes, elle est simple.

5.3.2. *Distinguer sous-classes et sous-types*

Les approches majoritaires des langages à objets — C++, JAVA mais aussi EIFFEL — reposent sur une identification des classes à des types faisant commuter le diagramme (5). Spécialisation et sous-typage doivent alors coïncider. Plusieurs auteurs ont cherché à résoudre les problèmes rencontrés en séparant spécialisation de classes et sous-typage : c'est le cas des langages DEE [GRO 91] et SATHER. L'argument des partisans de cette distinction est que, dans les langages classiques à la C++, EIFFEL ou JAVA, la relation de spécialisation de classes est « surchargée ». D'après eux, il faut distinguer ce qui relève de l'héritage de code et ce qui relève de la substituabilité. Aux caractéristiques principales du modèle standard (cf. section 2.1), il faut en effet rajouter la notion d'*encapsulation*, qui serait centrale suivant certains auteurs, en ce qu'elle permet de cacher l'implémentation de l'objet — typiquement ses attributs — ce qui fait de la classe l'implémentation d'un *type abstrait de données*. On peut donc considérer que la classe est constituée de trois éléments : son *type abstrait* (ce qui encapsule), son *type concret* (ce qui est encapsulé) et son code. C'est ce qui permet de considérer que l'héritage est « surchargé », puisqu'il recouvre à la fois le sous-typage (abstrait) et l'héritage d'implémentation (sous-typage concret et héritage de code).

L'objectif de ces approches va donc être de séparer la spécialisation de classes du polymorphisme, c'est-à-dire de la substituabilité. Dans un tel contexte, on peut se demander s'il doit y avoir une relation privilégiée entre les types d'une classe et de sa sous-classe : le sous-typage étant écarté, faut-il imposer le *matching* <# ou le sous-typage covariant <! ? En fait, à moins de vouloir faire coïncider la spécialisation de classes et la relation qui supporte la généricité bornée, dont on a vu que seul le *matching* la rendait sûre, aucune relation ne s'impose car il n'y a aucun polymorphisme entre deux classes. Or c'est le polymorphisme qui impose une substituabilité.

5.3.3. *Le langage* SATHER

A l'origine dérivé d'EIFFEL, le langage SATHER [SZY 94, OMO 95] a fini par séparer l'héritage du sous-typage. Ses auteurs le décrivent comme un langage contravariant, mais il est interprété par [WEB 92] comme un langage covariant mais pas polymorphe. Il s'avère que SATHER accepte une spécialisation covariante mais exige un sous-typage contravariant. Les classes sont les feuilles de la hiérarchie de types : deux classes ne sont donc jamais en relation de sous-typage, mais leurs types peuvent l'être, ou pas. La relation qui lie une classe à sa super-classe est considérée comme une copie textuelle. Dire que la spécialisation de classe est covariante est en fait impropre :

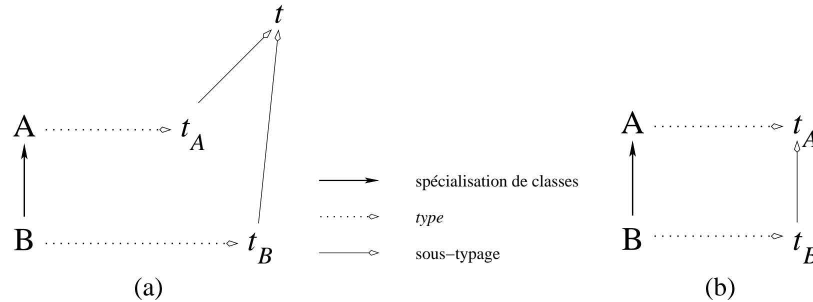

Figure 6 – Classes et types en SATHER : covariance (a) ou invariance (b)

en réalité, aucune règle ne contraint la redéfinition. Comme en JAVA ou C++, dès lors que la conformité des signatures n'est pas assurée, ce n'est même pas une redéfinition mais une surcharge statique : la méthode héritée coexiste avec la nouvelle définition.

L'impact de cette distinction sur le modèle covariant résultant de l'analyse est très fort. Supposons (figure 6-a) que l'on définisse une classe B, sous-classe d'une classe A, de types respectifs t_B et t_A. Les instances de A et de B ne peuvent être utilisées les unes à la place des autres que dans une expression typée par un super-type t commun à t_A et t_B et ne comportant aucune des propriétés de A que B redéfinit de façon covariante. Pour accéder aux autres propriétés, il faut avoir recours au *casting*, sous la forme de `typecase`, en prévoyant forcément un cas pour A et un cas pour B puisque les types des propriétés ne sont pas compatibles. L'argument du *casting* s'est donc déplacé : au lieu de le faire sur les paramètres des méthodes de B, dans la méthode appelée, on va le faire sur le receveur, instance de A ou de B, dans la méthode appelante. De plus, deux classes en relation de spécialisation invariante ne sont pas en relation de sous-typage. Mais il reste possible d'avoir le comportement polymorphe usuel, en typant par le type de la classe et non par la classe elle-même (figure 6-b). Cela impose juste une discipline de programmation consistant à expliciter le type des classes. Mais l'originalité de SATHER, c'est-à-dire la réalité de la distinction entre sous-typage et spécialisation, est finalement assez faible. En effet, comme toute redéfinition non invariante est traitée comme de la surcharge statique, le type de la sous-classe reste, en général, un sous-type du type de la super-classe. Apparemment, la seule exception provient d'une redéfinition non covariante du type de retour.

5.3.4. *Variation sur le polymorphisme d'inclusion : le langage* LOOM

Le polymorphisme d'inclusion peut être vu comme le fait qu'une expression d'un type statique τ_s soit lié à une valeur d'un type dynamique τ_d vérifiant une certaine relation de conformité avec le type statique. Usuellement, dans un système de types sûr, c'est le sous-typage basé sur la règle de contravariance, $\tau_d <: \tau_s$, et, en EIFFEL, ce sera le sous-typage covariant $\tau_d <! \tau_s$, avec les erreurs de types qu'il entraîne.

Plutôt que de chercher à faire coexister sous-typage et *matching*, dont on a vu que ce sont les relations de conformité les plus larges assurant un polymorphisme

sûr, qu'il soit d'inclusion ou paramétrique, le langage LOOM [BRU 97] a choisi le *matching* comme relation entre types : $\tau_d <\!\# \tau_s$. L'originalité du langage est d'obtenir un typage sûr pour un polymorphisme basé sur le *matching*.

Le typage sûr est obtenu au moyen de deux constructions de type : les types exacts n'autorisent aucun polymorphisme ($\tau_d = \tau_s$) alors que les types polymorphes, notés $\#t$ pour un type t, sont basés sur le *matching* ($\tau_d <\!\# \tau_s$). Ils sont obtenus par transformation du polymorphisme d'inclusion en polymorphisme paramétrique : une procédure `proc(x : #X)` est traitée comme une procédure générique `proc(T <# X, x : T)`. Le polymorphisme s'obtient par un équivalent de la règle de subsumption (4) :

$$\frac{u <\!\# v \qquad x : \#u}{x : \#v} \tag{12}$$

Comme un typage sûr est impossible avec le *matching* sans quelques restrictions, une règle, analogue à celle des *catcalls* polymorphes d'EIFFEL, interdit d'appliquer une méthode ayant un paramètre de type `mytype` à un receveur dont le type statique n'est pas exact, c'est-à-dire dont le type dynamique n'est pas connu statiquement.

5.4. *Spécialisation ou généricité ?*

La spécialisation et la généricité constituent les deux grandes approches de la réutilisation et de la factorisation de code. Si la spécialisation est propre à l'approche objet, la généricité a des origines plus lointaines : CLU, ADA, etc. Pourtant, spécialisation et généricité correspondent respectivement aux polymorphismes d'inclusion et paramétriques, dont on a vu qu'ils n'étaient pas si différents que cela, au fond : le premier ne s'exprime pleinement que par l'intermédiaire du second. Les propositions pour rendre sûr le système de types d'EIFFEL [COO 89, WEB 92], de même que le langage LOOM [BRU 97], consistent à remplacer les redéfinitions non sûres par des paramètres de types. Mais une telle approche est aussi possible au niveau de la conception : un schéma de conception très général consisterait à traduire un modèle d'analyse covariant en modèle d'implémentation basé sur la généricité.

5.4.1. *Un exemple : les graphes d'objets*

Nous voulons définir des `graphes` d'objets, dont les `sommets` et les `arêtes` sont des objets. Ces graphes sont spécialisés par deux exemples spécifiques : d'un côté des `molécules` en chimie, dont les sommets sont des `atomes` et les arêtes des `liaisons` ; de l'autre, des `réseaux` de télécommunications, dont les sommets sont des `nœuds` et les arêtes des `canaux`. La figure 7 donne un schéma de modélisation naturel, basé sur une spécialisation covariante : chacune des propriétés figurant dans le schéma pourrait servir d'ancre pour typer toutes les autres propriétés non citées.

On voit bien comment les problèmes soulevés par le conflit entre spécialisation covariante et substituabilité contravariante se traduiraient dans cet exemple. Avec la contravariance, une méthode `ajoutSommet` permettrait d'ajouter un `Atome` à un `Réseau`, ou un `Canal` à une `Molécule`, ce qui traduit un certain désordre. Toute autre

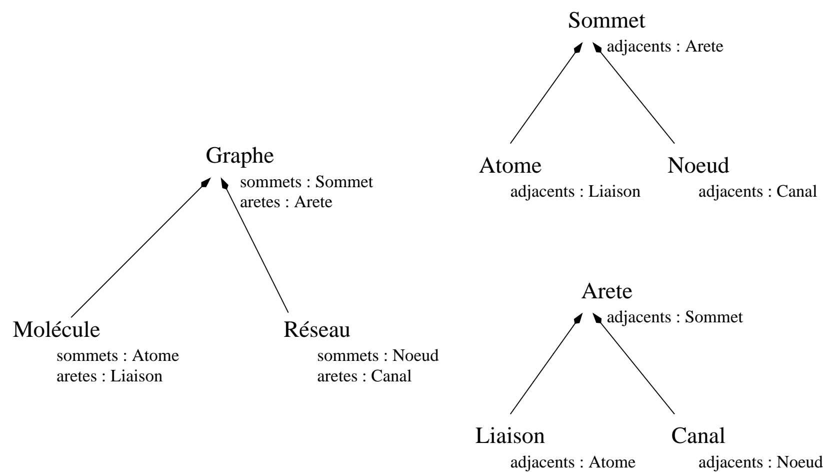

Figure 7 – Spécialisation covariante de graphes d'objets.

solution, telle que spécialisation covariante, sélection multiple ou *casting* conduit à des erreurs de type. Pour assurer la bonne réutilisation et extensibilité du code, il est tentant d'essayer la généricité, en modélisant les graphes par une classe paramétrée :

```
Graphe⟨U,V⟩
      sommets:U
      aretes:V
      ajoutSommet(U)
```

Les molécules sont alors représentées par `Graphe⟨Atome,Liaison⟩` et les réseaux par `Graphe⟨Nœud,Canal⟩`, qui sont obtenus par substitution des types formels `U` et `V`. Tant que l'on ne considère que la classe `Graphe`, et en acceptant la perte de la spécialisation, la généricité donne un bon résultat en ce qu'elle évite toute redondance de code, tout en permettant de typer correctement `sommets` et `arêtes`. Les choses se gâtent dès que l'on considère les classes en relation. Ainsi, pour éviter des chimères comme `Graphe⟨Atome,Canal⟩`, il faudrait faire de `Sommet` et `Arête` des classes paramétrées, respectivement, par l'`Arête` et le `Sommet` adjacents :

```
Sommet⟨V⟩                    Arête⟨U⟩
      adjacents:V                  adjacents:U
```

Le problème, apparemment plus pratique que théorique, est que cette définition implique une récursivité croisée qui n'est pas admise par tous les langages. [BRU 98] montre, sur un exemple voisin, qu'une telle récursivité est possible en PIZZA [ODE 97], bien que son expression soit complexe. Les auteurs constatent que les types virtuels, qui sont très proches des types ancrés, simplifient énormément cette expression mais en introduisant des erreurs de types. Les clauses `where` autorisent aussi une expression équivalente [DAY 95]. Une solution générale, qui manque aux langages considérés, serait de pouvoir exprimer directement le système d'équations de types suivant :

```
Atome = Sommet⟨Liaison⟩        Liaison = Arête⟨Atome⟩
```

Nonobstant ce problème de circularité, la généricité est donc une bonne solution pour le typage covariant et sûr « à l'intérieur » du type `graphe`. Mais le typage sûr interdit alors la substitution d'une `Molécule` à un `Graphe`. Le problème se pose ainsi pour les classes « extérieures » au graphe d'objets, par exemple pour un visualisateur de graphes qui devrait forcément être lui aussi paramétré si l'on veut qu'il puisse visualiser n'importe quel type de graphe, surtout si ses fonctionnalités dépendent du type de graphe visualisé : on ne visualise pas une molécule comme un réseau de télécommunications. Une modélisation en SATHER conduirait au même obstacle, la spécialisation covariante des `Graphes` par des `Molécules` interdisant le polymorphisme.

5.4.2. *La généricité n'est pas une spécialisation*

L'utilisation de la généricité pour répondre à des besoins de spécialisation est donc possible, mais elle ne fait que déplacer les problèmes tout en entraînant une confusion conceptuelle dangereuse. Un type paramétré est souvent présenté de façon fonctionnelle, comme une fonction de `Types` → `Types`. Dans un contexte objet, il serait pourtant préférable de le considérer comme une classe dont les instances sont des classes : une méta-classe donc. Conceptuellement, spécialisation et généricité peuvent alors se voir comme les deux relations fondamentales de l'approche objet, qu'il serait dangereux de confondre. Cette confusion a accompagné toute l'évolution des objets depuis leurs débuts. Elle est à la base des langages de prototypes et, bien que condamnée très tôt dans le domaine de la représentation des connaissances [BRA 83], elle resurgit périodiquement dans la technique des *mixins*. Les *mixins* ont été originellement proposés dans les langages à typage dynamique (FLAVORS, LOOPS) en prétendant répondre aux problèmes causés par l'héritage multiple. Dans leur version plus moderne, ils sont définis comme des classes paramétrées par la super-classe qu'elles étendent (*parametric heir classes*). [ANC 00] propose ainsi une extension de JAVA à l'héritage multiple : au lieu de définir par héritage multiple une classe A comme spécialisation d'une classe B et d'un *mixin* M, on définira A comme (sous-classe de) $M\langle B\rangle$, ce qui est désigné explicitement par les auteurs comme une « instanciation ». Dans cette logique, [BRA 90] propose une vision de l'héritage où chaque sous-classe serait un *mixin* paramétré par sa super-classe, ce qui rejoint la substitution de types de [PAL 90].

6. Compilation séparée et analyse globale des types

Les systèmes de types que nous avons examinés ont pour propriété principale d'être insensibles au contexte, en ce sens que la correction d'une expression ne dépend que des types statiques de ses sous-expressions. C'est une restriction particulièrement pessimiste du principe de substituabilité, qui se justifie par la simplicité de sa mise en œuvre, notamment en compilation séparée. A l'opposé, la version la plus optimiste consisterait à examiner toutes les exécutions possibles, par une sorte d'interprétation abstraite, en l'occurrence une analyse statique des types. Une telle analyse de types consistera à déterminer, pour chaque expression d'une certaine catégorie — par exemple les receveurs ou les appels de méthodes —, l'ensemble des types effectifs que pourra prendre la valeur de l'expression à l'exécution. On remplace ainsi le

type polymorphe explicite du programme par un ensemble (une disjonction) de types exacts. Une analyse exacte est impossible, le problème étant clairement indécidable, mais de nombreuses heuristiques fournissent une borne supérieure suffisante [GIL 98]. L'analyse de types a alors deux fonctions principales : détecter les erreurs de types et distinguer le code vivant du code mort pour améliorer l'efficacité du code généré. L'inconvénient majeur de l'analyse de types réside ainsi dans ce qu'elle nécessite la connaissance de l'ensemble des types utilisé par un programme : la hiérarchie doit être *fermée*. L'analyse de types est donc difficilement compatible avec la compilation séparée. Qu'un système de types nécessite une analyse globale, est donc un obstacle à la réutilisation, la définition d'une nouvelle classe ne pouvant plus être considérée comme une « extension conservative »[4] de l'ensemble de classes pré-existant.

La règle de contravariance fournit un critère de substituabilité adapté à la compilation séparée. Dès que l'on s'en écarte, par exemple avec la covariance, une analyse de types est souhaitable pour limiter les erreurs de types ou les vérifications dynamiques. La règle des *catcalls* d'EIFFEL repose ainsi sur une analyse de types rudimentaire, implicite dans la définition même du *catcall*. Un traitement moins grossier, ne serait-ce qu'une définition plus précise d'un appel polymorphe, nécessite une analyse de types beaucoup plus fine, puisqu'il faut typer chaque appel de méthode dans le produit des types du receveur et des paramètres [ROY 02]. En revanche, on peut caractériser le système de types de LOOM comme une réponse au problème des *catcalls* compatible avec une compilation séparée. Dès que l'on accepte la moindre analyse de types, il est possible de s'affranchir du cadre minimal de typage que nous avons adopté pour l'appel de méthode, à savoir que le type statique du receveur ait une méthode du même nom. On peut alors adopter l'approche de la *rétraction*, ou coercition descendante implicite, empruntée au cadre des spécifications algébriques, qui repose sur l'existence d'un unique sous-type qui introduit la méthode en question [ROY 02]. L'approche du typage sûr peut même s'inverser, jusqu'à accepter une expression tant qu'il n'est pas prouvé que toutes ses exécutions entraîneront une erreur. C'est la démarche suivie par le langage CLAIRE [JOS 01]. Notons, pour finir, qu'il ne faut pas confondre cette analyse de types avec l'inférence de types à la ML, mise en œuvre dans OBJECTIVE-CAML, dont l'objectif est de permettre un typage statique orthodoxe sans nécessiter d'annotations de types et qui reste compatible avec une compilation séparée.

7. Propositions pour un langage covariant

A l'issue de ce tour d'horizon de différents systèmes de types, nous n'en avons trouvé aucun qui soit compatible avec un modèle d'analyse covariant. Le lecteur n'en sera pas étonné outre mesure : les erreurs de types ayant été proclamées partie intégrante du modèle d'analyse, il faut bien qu'elles perdurent jusqu'à l'implémentation. Les langages standard en témoignent, qui, grâce au *casting*, réintroduisent les erreurs

4. F est une extension conservative de E si F contient E et que toute déduction faite dans F, ($x \vdash_F y$) dont les antécédents (x) et la conclusion (y) sont dans E, peut être faite dans E ($x \vdash_E y$). Ce que l'on pourrait formaliser par : $E \subset F, x, y \in E, x \vdash_F y \Rightarrow x \vdash_E y$.

de types dans des systèmes prétendûment sûrs. Un système de types sûr est impossible en pratique et le problème est donc légèrement décalé : il s'agit de trouver un système de types aussi sûr que possible, intégrant et gérant, jusqu'à l'exécution, les erreurs de types potentielles. De plus, ce système de types doit être aussi simple et direct que possible : dans le cadre du typage sûr, la coexistence de plusieurs relations entre types — sous-typage, *matching*, types exacts — ou entre types et classes, atteint une complexité inaccessible à la majorité des développeurs. Même s'il est basé sur des théories plus complexes, le système de types idéal devra donc fournir, au niveau utilisateur, un ensemble de concepts très limités : le modèle classique identifiant les classes à des types et la spécialisation à du sous-typage est de loin préférable.

Au terme de cette analyse, deux approches nous paraissent également éligibles : un langage covariant en sélection simple, à la EIFFEL, ou un langage en sélection multiple autorisant un traitement covariant de la redéfinition.

Un langage covariant en sélection simple est certainement la réponse qui correspond le mieux à la question posée : dans le modèle objet majoritaire de la sélection simple, la covariance permet de se passer très largement de la coercition et de localiser parfaitement les erreurs de types là où le modèle d'analyse les impose. La spécification détaillée d'un tel langage n'est pas notre objectif ici. Précisons néanmoins quelques points :

– pas plus de surcharge statique que de coercition, à l'exclusion d'une construction générale à branchement multiple comme `typecase` ;

– un système de types incluant `mytype` et des types ancrés, éventuellement sous la forme syntaxique des types virtuels ou sous une forme à trouver, et étendus à des types ancrés sur des chemins ou des types formels ;

– la présence de la généricité bornée devrait être fortement mise en question : l'instanciation automatique d'un type paramétré $A\langle C\rangle$ pourrait être remplacée par la spécialisation automatique de types ancrés ou virtuels, par exemple par une construction syntaxique comme A `with` $p : C$, où p serait l'ancre correspondant au type formel ;

– un système de types distinguant les erreurs imposant une correction, des erreurs potentielles dues à la covariance, qui se contentent d'une vérification dynamique ;

– pour ces dernières erreurs, un signalement explicite, à la compilation, pour que le programmeur puisse les confronter au modèle d'analyse initial, et à l'exécution, pour que leur gestion fasse partie de la conception des applications.

Le langage EIFFEL est un peu omniprésent dans cette analyse et dans ses conclusions. Ce n'est pas que l'auteur en soit un chaud partisan mais il est inévitable, étant le seul langage covariant largement connu et reconnu. Comparé au désastre conceptuel que constitue C++, il faut reconnaître à EIFFEL la qualité de beaucoup de ses concepts, avec des exceptions comme les types expansés ou l'héritage répété, qui répètent quelques erreurs de C++. On critiquera, en revanche, la complexité du langage, dont les multiples concepts se recouvrent passablement — on l'a vu pour la généricité et les types ancrés — et dont la syntaxe est souvent verbeuse. Son système de types mériterait d'être étudié plus formellement, non pas pour le ramener à l'orthodoxie de

la contravariance comme ont pu le faire [COO 89, WEB 92] mais pour améliorer encore son expressivité (formulation symétrique, chemins comme ancres, rapport avec les types virtuels etc.) et pour y faire figurer expressément les erreurs de types.

La sélection multiple, dans sa variante avec multi-méthodes, permet de simuler simplement la redéfinition covariante des paramètres de méthodes. Cela peut donc être une technique d'implémentation d'un langage covariant en sélection simple, mais il n'est pas sûr que ce soit un choix d'implémentation judicieux. En revanche, dans le cadre d'un langage avec sélection multiple et multi-méthodes, la technique de [BOY 96] permet de générer les branches manquantes en leur faisant signaler une exception. La coexistence de la sélection multiple avec la redéfinition covariante est ainsi possible : on peut même faire de la multi-redéfinition covariante. Par rapport au système de types des langages en sélection multiple, il serait profitable de rajouter un équivalent de `mytype` et des types ancrés. Enfin, la sélection multiple a l'avantage d'éviter tout recours à la coercition de types. Le traitement covariant des attributs devra être assuré de la même manière qu'en sélection simple.

Dans les deux cas, la possibilité de la covariance ne doit pas en faire une obligation : un style purement invariant reste possible voire souhaitable dans de nombreux cas. Le langage peut donc introduire des annotations au niveau des classes ou des propriétés pour limiter la redéfinition, par sous-typage strict, voire *matching*, soit d'une propriété, soit d'un type : dans ce dernier cas, la restriction s'étend à chaque propriété du type. Ces spécifications ne vont pas jusqu'à décider d'une compilation séparée ou globale. En cas de compilation séparée, le système de types doit être suffisamment optimiste pour accepter toute définition de classes qui peut donner lieu à des exécutions valides. L'idéal serait d'allier la compilation séparée et l'analyse globale de types, selon le schéma proposé par [PRI 02], en faisant précéder l'édition de liens d'une analyse globale à partir des analyses locales des flux de types effectuées sur chaque classe. En amont, la conception devra intégrer les risques d'erreurs de type liées à la spécialisation covariante, sous la forme d'une gestion adéquate des exceptions associées.

8. Conclusions et perspectives

Notre analyse du problème peut se résumer en trois points : 1) la modélisation objet, telle qu'elle est produite par une phase d'analyse, relève d'un modèle de spécialisation conceptuelle qui est intrinsèquement covariante ; 2) la théorie des types a montré, irréfutablement, que le typage sûr n'est possible qu'avec la contravariance, au moins dans un cadre de compilation séparée ; 3) nous n'avons rencontré aucune alternative satisfaisante aux langages covariants (qui peuvent par ailleurs s'améliorer). Les langages standards (C++, JAVA) ont une politique invariante — orthodoxe pour C++, trop restrictive pour JAVA — mais ils réintroduisent les erreurs de type, de façon incontrôlée, avec le *casting*, et leur surcharge statique rend incompréhensible la règle de contravariance. Les solutions basées sur la généricité, sur le *matching* ou sur la différenciation entre spécialisation et sous-typage éloignent beaucoup du modèle d'analyse : elles reviennent toutes à échanger le polymorphisme contre la covariance.

Notre conclusion est catégorique. Les erreurs de type sont inhérentes à la spécialisation : il faut donc les prendre en compte dans les langages et, en amont, dans les méthodes d'analyse et de conception. S'il ne faut pas forcément appliquer la règle de contravariance, il faut bien la connaître pour appréhender correctement, à toutes les étapes, les risques liés à la covariance. Il faut enfin concevoir des langages qui assurent le meilleur compromis entre la *sûreté* du typage et l'expressivité du langage, c'est-à-dire sa proximité de la sémantique naturelle de la spécialisation.

Au terme de cette analyse, on peut donc critiquer les langages de programmation pour avoir suivi trop aveuglément le dogme du typage sûr alors qu'il aurait été plus raisonnable, à l'inverse, de chercher à se mettre au service des modèles d'analyse. Mais la faute est assez partagée et la situation révèle plutôt une certaine démission de la part des méthodes d'analyse et de conception. Alors qu'elles font la part belle à la spécialisation pour structurer les modèles, voire comme structure conceptuelle essentielle, bien peu approfondissent la sémantique de cette spécialisation. En particulier, la spécification du rôle et de la sémantique de la redéfinition est absente de la plupart des méthodes : quand elles le font, c'est pour s'aligner sur les (certains) langages, par exemple en imposant l'invariance des signatures [RUM 91]. De même, si les méthodes sont souvent prolixes pour ce qui est des conventions de nommage pour les différentes entités d'un modèle ou d'un programme, elles gardent un silence prudent sur la surcharge. Ces lacunes s'expliquent sans doute en partie par le fait que les méthodes se reposent sur les langages pour imposer leurs propres spécifications. Il est significatif que l'analyse que nous en avons faite s'appuie sur des travaux relevant de la représentation des connaissances et non du génie logiciel.

Trois perspectives principales s'ouvrent, qui permettraient d'améliorer notre argumentation : la formalisation d'un système de types intégrant l'erreur de type liée à la covariance, une implémentation et la prise en compte de nos critiques dans la métrologie des logiciels. Une implémentation s'impose, par exemple en se basant sur un langage et un compilateur existant, en en pervertissant subtilement les règles : JAVA et EIFFEL sont des cibles naturelles, mais pas C++, dont le cas est, à tous égards, désespéré. La compilation reste ainsi un axe de recherche d'actualité, avec deux thèmes principaux : les inférences de types, pour améliorer les prévisions du compilateur et la réconciliation des compilations globale et séparée, pour optimiser avec la première et distribuer grâce à la seconde. Enfin, la métrologie pourrait apporter un argument en évaluant le coût de la contravariance dans les applications. Ce coût devrait mesurer les différentes contorsions nécessaires à compenser la perte de la covariance, au premier rang desquelles le *casting*. [WAN 01] fournit de premiers chiffres, encore insuffisants. En contrepartie, il faudra pénaliser, dans un langage covariant, les *catcalls* polymorphes dont il n'est pas prouvé que toutes les exécutions sont sûres.

Remerciements

L'auteur remercie Michel Dao de lui avoir signalé l'existence de la technique de simulation de la covariance en JAVA, Marianne Huchard pour son expertise en JAVA et C++, Jean-Claude Royer et les relecteurs anonymes pour leurs suggestions avisées.

9. Bibliographie

[ABA 95] ABADI M., CARDELLI L., « On Subtyping and Matching », OLTHOFF W., Ed., *Proc. ECOOP'95*, LNCS 952, Springer-Verlag, 1995, p. 145–167.

[ABI 95] ABITEBOUL S., HULL R., VIANU V., *Foundations of Databases*, Addison-Wesley, Reading (MA), USA, 1995.

[AMA 93] AMADIO R., CARDELLI L., « Subtyping recursive types », *ACM Transactions on Programming Languages and Systems*, vol. 15, n° 4, 1993, p. 575–631.

[ANC 00] ANCONA D., LAGORIO G., ZUCCA E., « JAM — A Smooth Extension of Java with Mixins », BERTINO E., Ed., *Proc. ECOOP'2000*, LNCS 1850, Springer-Verlag, 2000, p. 154–178.

[App95] Apple Computer, « DYLAN reference manual », 1995.

[ARN 97] ARNOLD K., GOSLING J., *The JAVA programming language, Second edition*, Addison-Wesley, 1997.

[BEU 02] BEUGNARD A., « Une comparaison de langages objet relative au traitement de la redéfinition de méthode et à la liaison dynamique », DAO M., HUCHARD M., Eds., *Actes LMO'2002* in *L'Objet vol. 8*, Hermès, 2002, p. 99–114.

[BOB 88] BOBROW D., DEMICHIEL L., GABRIEL R., KEENE S., KICZALES G., MOON D., « Common Lisp Object System Specification, », *ACM SIGPLAN Notices*, vol. 23, 1988.

[BOO 94] BOOCH G., *Object Oriented Analysis and Design with Applications*, Benjamin Cummings Publishing, Reading (MA), USA, second édition, 1994.

[BOY 96] BOYLAND J., CASTAGNA G., « Type-safe compilation of covariant specialization : a practical case », COINTE P., Ed., *Proc. ECOOP'96*, LNCS 1098, Springer-Verlag, 1996, p. 3–25.

[BRA 83] BRACHMAN R., « What IS-A is and isn't : An Analysis of Taxonomic Links in Semantic Networks », *IEEE Computer*, vol. 16, n° 10, 1983, p. 30–37.

[BRA 90] BRACHA G., COOK W., « Mixin-based Inheritance », *Proc. OOPSLA/ECOOP'90*, SIGPLAN Notices, 25(10), ACM Press, 1990, p. 303–311.

[BRU 95] BRUCE K. B., A.SCHUETT, VAN GENT R., « PolyTOIL : A Type-Safe Polymorphic Object-Oriented Language », OLTHOFF W., Ed., *Proc. ECOOP'95*, LNCS 952, Springer-Verlag, 1995, p. 27–51.

[BRU 97] BRUCE K. B., PETERSEN L., FIECH A., « Subtyping Is Not a Good "Match" for Object-Oriented Languages », AKSIT M., MATSUOKA S., Eds., *Proc. ECOOP'97*, LNCS 1241, Springer-Verlag, 1997, p. 104–127.

[BRU 98] BRUCE K. B., ODERSKY M., WADLER P., « A Statically Safe Alternative to Virtual Types », JUL E., Ed., *Proc. ECOOP'98*, LNCS 1445, Springer-Verlag, 1998, p. 523–549.

[CAN 89] CANNING P., COOK W., HILL W., MITCHELL J., OLTHOFF W., « F-bounded polymorphism for object-oriented programming », *Proc. ACM Conf. Functional Programming and Computer Architecture*, 1989, p. 273-280.

[CAR 84] CARDELLI L., « A Semantics of Multiple Inheritance », KAHN G., MCQUEEN D., PLOTKIN G., Eds., *Semantics of Data Types*, LNCS 173, p. 51–67, Springer-Verlag, Berlin, 1984.

[CAR 85] CARDELLI L., WEGNER P., « On understanding types, data abstraction and polymorphism », *ACM Computing Surveys*, vol. 17, n° 4, 1985.

[CAR 93] CARGILL C., « Aggregation : Inheritance vs Member Objects », *C++ Report*, vol. 5, n° 1, 1993.

[CAS 95] CASTAGNA G., « Covariance and Contravariance : Conflict Without a Cause », *ACM Transactions on Programming Languages and Systems*, vol. 17, n° 3, 1995, p. 431–437.

[CAS 96a] CASTAGNA G., « Instance variables specialization in object-oriented programming », (unpublished), 1996.

[CAS 96b] CASTAGNA G., « Le modèle fondé sur la surcharge : une visite guidée », *Technique et Science Informatiques*, vol. 15, n° 6, 1996, p. 673–708.

[CAS 97] CASTAGNA G., *Object-oriented programming : a unified foundation*, Progress in Theoretical Computer Science Series, Birkhaüser, 1997.

[CAS 99] CASEAU Y., LABURTHE F., « Introduction to the CLAIRE programming language », rapport, 1999, LIENS.

[CHA 93] CHAMBERS C., « The CECIL Language, Specification and Rationale », Technical Report n° 93-03-05, 1993, University of Washington.

[CHI 95] CHIBA S., « A Metaobject Protocol for C++ », *Proc. OOPSLA'95*, SIGPLAN Notices, 30(10), ACM Press, 1995, p. 285–299.

[COO 89] COOK W. R., « A Proposal for Making Eiffel Type-safe », COOK S., Ed., *Proc. ECOOP'89*, Cambridge University Press, 1989, p. 57–70.

[COO 90] COOK W., HILL W., CANNING P., « Inheritance Is Not Subtyping », *Proc. POPL'90*, ACM Press, 1990, p. 125–135.

[DAY 95] DAY M., GRUBER R., LISKOV B., MYERS A., « Subtypes vs. Where clauses. Constraining Parametric Polymorphism », *Proc. OOPSLA'95*, ACM Press, 1995, p. 156–168.

[DUC 89] DUCOURNAU R., HABIB M., « La multiplicité de l'héritage dans les langages à objets », *Technique et Science Informatiques*, vol. 8, n° 1, 1989, p. 41–62.

[DUC 95] DUCOURNAU R., HABIB M., HUCHARD M., MUGNIER M.-L., NAPOLI A., « Le point sur l'héritage multiple », *Technique et Science Informatiques*, vol. 14, n° 3, 1995, p. 309–345, Hermès.

[DUC 96] DUCOURNAU R., « Des langages à objets aux logiques terminologiques : les systèmes classificatoires », Rapport de Recherche n° 96-030, 1996, L.I.R.M.M.

[DUC 98] DUCOURNAU R., « La logique des objets : application à la classification incertaine », DUCOURNAU R., EUZENAT J., MASINI G., NAPOLI A., Eds., *Langages et modèles à objets*, Collection Didactique, chapitre 12, p. 351–380, INRIA, 1998.

[DUC 01a] DUCOURNAU R., « La compilation de l'envoi de message dans les langages statiques », Rapport de Recherche n° 01-014, 2001, L.I.R.M.M.

[DUC 01b] DUCOURNAU R., PAVILLET G., « Langage à objets et logique de descriptions : un schéma d'intégration », BORNE I., GODIN R., Eds., *Actes LMO'2001* in *L'Objet vol. 7*, Hermès, 2001, p. 233–249.

[ELL 90] ELLIS M., STROUSTRUP B., *The annotated C++ reference manual*, Addison-Wesley, Reading, MA, US, 1990.

[GHE 91] GHELLI G., « A Static Type System for Message Passing », *Proc. OOPSLA'91*, SIGPLAN Notices, 26(10), ACM Press, 1991, p. 129–145.

[GIL 98] GIL J., ITAI A., « The Complexity of Type Analysis of Object Oriented Programs », *Proc. ECOOP'98*, LNCS 1445, Springer-Verlag, 1998, p. 601–634.

[GOD 95] GODIN R., MINEAU G., MISSAOUI R., MILI H., « Méthodes de classification conceptuelle basées sur les treillis de Galois et applications », *Revue d'Intelligence Artificielle*, vol. 9, n° 2, 1995, p. 105–137.

[GOL 83] GOLDBERG A., ROBSON D., SMALLTALK *: the language and its implementation*, Addison-Wesley, Reading, MA, 1983.

[GRO 91] GROGONO P., « Issues in the design of an Object-Oriented Programming Language », *Structured Programming*, vol. 12, n° 1, 1991, p. 1–16.

[HAB 97] HABIB M., NOURINE L., RAYNAUD O., « A new lattice-based heuristic for taxonomy encoding », *Proc. KRUSE'97*, 1997, p. 60–71.

[HUC 00] HUCHARD M., LEBLANC H., « Computing Interfaces in Java », *Proc. of IEEE Int. Conf. on Automated Software Engineering (ASE'2000)*, 2000, p. 317–320.

[JOH 88] JOHNSON R., FOOTE B., « Designing reusable classes », *Journal of Object-Oriented Programming*, vol. 1, n° 2, 1988, p. 22–35.

[JOS 01] JOSSET F.-X., CASEAU Y., « Optimisation de code dans le langage CLAIRE », *Actes LMO'2001* in *L'Objet vol. 7*, Hermès, 2001.

[MAS 89] MASINI G., NAPOLI A., COLNET D., LÉONARD D., TOMBRE K., *Les langages à objets*, InterEditions, Paris, 1989.

[MEY 92] MEYER B., *Eiffel : The Language*, Prentice Hall Object-Oriented Series, Prentice Hall International, Hemel Hempstead, UK, 1992.

[MEY 97] MEYER B., *Object-Oriented Software Construction*, The Object-Oriented Series, Prentice-Hall, Englewood Cliffs (NJ), USA, second édition, 1997.

[MUG 91] MUGRIDGE W. B., HAMER J., HOSKING J. G., « Multi-Methods in a Statically-Typed Programming Language », AMERICA P., Ed., *Proc. ECOOP'91*, LNCS 512, Springer-Verlag, 1991, p. 307–324.

[ODE 97] ODERSKY M., WADLER P., « Pizza into Java : Translating Theory into Practice », *Proc. POPL'97*, ACM Press, 1997, p. 146–159.

[OMO 95] OMOHUNDRO S., STOUTAMIRE D., « The Sather 1.0 Specification », rapport n° TR-95-057, 1995, Int. Computer Science Institute, Berkeley (CA), USA.

[PAL 90] PALSBERG J., SCHWARTZBACH M. J., « Type Substitution for Object-Oriented Programming », *Proc. OOPSLA/ECOOP'90*, SIGPLAN Notices, 25(10), ACM Press, 1990, p. 151–160.

[PER 98] PERROT J.-F., « Objets, classes, héritage : définitions », DUCOURNAU R., EUZENAT J., MASINI G., NAPOLI A., Eds., *Langages et modèles à objets*, Collection Didactique, chapitre 1, p. 3–34, INRIA, 1998.

[PLA 98] PLAINDOUX D., BODEVEIX J.-P., PERCEBOIS C., « Types versus classes », *L'Objet*, vol. 4, n° 1, 1998, p. 9–44, Hermès.

[PRI 02] PRIVAT J., « Analyse de types et graphe d'appels en compilation séparée », Mémoire de DEA, Université Montpellier II, 2002.

[PéT 97] PÉTRA, GROUPE DE TRAVAIL, « Modèle de conception de réseaux commutés pour l'ingénierie du trafic », rapport n° DE/DAC/GTR/49.97, décembre 1997, CNET.

[RAY 00] RAYSIDE D., CAMPBELL G., « An Aristotelian Understanding of Object-Oriented Programming », *Proc. OOPSLA'00*, SIGPLAN Notices, 35(10), ACM Press, 2000, p. 337–353.

[RAY 01] RAYNAUD O., THIERRY E., « A Quasi Optimal Bit-Vector Encoding of Tree Hierarchies. Application to Efficient Type Inclusion Tests », *Proc. ECOOP'2001*, LNCS 2072, Springer-Verlag, 2001, p. 165–180.

[RéM 97] RÉMY D., VOUILLON J., « Objective ML : A Simple Object-Oriented Extension of ML », *Proceedings of the 24th ACM Conference on Principles of Programming Languages (POPL'97), Paris, France*, 1997, p. 40–53.

[ROY 02] ROYER J.-C., « An Operational Approach to the Semantics of Classes : Application to Type Checking », *Programming and Computer Software*, vol. 28, n° 3, 2002, (to appear).

[RUM 91] RUMBAUGH J., BLAHA M., PREMERLANI W., EDDY F., LORENSEN W., *Object Oriented Modeling and Design*, Prentice-Hall, Englewood Cliffs (NJ), USA, 1991.

[STE 86] STEFIK M., BOBROW D., « Object-Oriented Programming : Themes and Variations », *AI Magazine*, vol. 6, n° 4, 1986, p. 40–62.

[STR 98] STROUSTRUP B., *The C++ programming Language, 3e ed.*, Addison-Wesley, 1998.

[SZY 94] SZYPERSKY C., OMOHUNDRO S., MURER S., « Engineering a Programming Language : The Type and Class System of Sather », *Proc. of First Int. Conference on Programming Languages and System Architectures*, LNCS 782, Springer Verlag, 1994.

[VIT 97] VITEK J., HORSPOOL R., KRALL A., « Efficient Type Inclusion Tests », *Proc. OOPSLA'97*, SIGPLAN Notices, 32(10), ACM Press, 1997, p. 142–157.

[WAN 89] WAND Y., « A proposal for a formal model of objects », KIM W., LOCHOVSKY F., Eds., *Object-Oriented Concepts, Databases and Applications*, p. 537–559, ACM Press, 1989.

[WAN 01] WANG T., SMITH S., « Precise Constraint-Based Type Inference for Java », *Proc. ECOOP'2001*, LNCS 2072, Springer-Verlag, 2001, p. 99–117.

[WEB 92] WEBER F., « Getting Class Correctness and System Correctness Equivalent — How to Get Covariant Right », EGE R., SINGH M., MEYER B., Eds., *Technology of Object-Oriented Languages and Systems (TOOLS 8)*, 1992, p. 192–213.

[WEG 86] WEGNER P., « Classification in Object-Oriented Systems », *ACM SIGPLAN Notices*, vol. 21, n° 10, 1986, p. 173–182.

[WEG 88] WEGNER P., ZDONIK S. R., « Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like », GJESSING S., NYGAARD K., Eds., *Proc. ECOOP'88*, LNCS 322, Springer-Verlag, 1988, p. 55–77.

[WIL 92] WILLE R., « Concept lattices and conceptual knowledge systems », *Computers Mth. Applic.*, vol. 23, n° 6-9, 1992, p. 493-515.

[WOO 92] WOODS W., SCHMOLZE J., « The KL-ONE Family », *Computers & Mathematics with Applications*, vol. 23, n° 2–5, 1992, p. 133–177.

***Roland Ducournau** est professeur à l'Université Montpellier II. Son domaine de recherche est « l'objet », sous toutes ses formes, mais surtout du point de vue de ses capacités de représentation : en représentation des connaissances, aussi bien qu'en programmation ou en bases de données. La structure classificatoire des systèmes d'objets est au cœur de ses travaux : sémantique de l'héritage et de la classification, implémentation de l'envoi de message, etc.*

Cet article a été initialement soumis
à la revue L'Objet intégrée à RSTI