Université Paris 7 - Denis Diderot
UFR d'informatique

Thèse

pour l'obtention du diplôme de

Docteur de l'université Paris 7
Spécialité : informatique

présentée et soutenue publiquement par

Jérôme VOUILLON

le 5 octobre 2000

Conception et réalisation d'une extension du langage ML avec des objets

Directeur de thèse

M. Michel Mauny

Jury

M. Gérard BERRY
M. Yves CASEAU
M. Pierre COINTE
M. Roberto DI COSMO
M. Martin ODERSKY
M. Benjamin PIERCE
M. Didier RÉMY

Université Paris 7 - Denis Diderot
UFR d'informatique

Thèse

pour l'obtention du diplôme de

Docteur de l'université Paris 7
Spécialité : informatique

présentée et soutenue publiquement par

Jérôme VOUILLON

le 5 octobre 2000

Conception et réalisation d'une extension du langage ML avec des objets

Directeur de thèse

M. Michel Mauny

Jury

M. Gérard BERRY
M. Yves CASEAU
M. Pierre COINTE
M. Roberto DI COSMO
M. Martin ODERSKY
M. Benjamin PIERCE
M. Didier RÉMY

Résumé

Les objets se sont imposés comme un concept informatique incontournable : les principaux langages de programmation utilisés actuellement ont soit été conçus à la base pour permettre ce style de programmation, soient ont été étendues avec les constructions nécessaires.

Le language ML est un langage fonctionnel statiquement typé avec synthèse de type. L'addition d'objets à ce langage s'avère difficile, notamment à cause des contraintes très fortes nécessaire afin de pouvoir synthétiser les types.

Ce travail présente une extension de ce langage avec des objets et des classes qui s'intègre bien avec les autre constructions de ML, tout en étant très expressive. Nous décrivons en détail l'ensemble de cette extension : nous présentons le typage des objets et des classes, et montrons comment ces contructions sont compilées.

Mots-clés

Langages de programmation. ML. Classes. Héritage. Objets. Systèmes de modules. Sémantique. Typage. Typage statique. Polymorphisme. Synthèse de type.

Adresse du laboratoire

Projet Cristal
INRIA Rocquencourt
Domaine de Voluceau
B.P. 105
78153 Le Chesnay

Remerciements

Je tiens à remercier Benjamin Pierce et Martin Odersky d'avoir accepté la lourde charge d'être rapporteurs de cette thèse. Je remercie également Gérard Berry, Yves Caseau, Pierre Cointe et Roberto Di Cosmo pour avoir bien voulu s'intéresser à ce travail.

Je tiens également à remercier Didier Rémy pour m'avoir guidé dans cette thèse, et tous les membres du projet Cristal pour leur aide aux cours de ces quatres années.

Enfin, merci à Xavier Leroy pour ce merveilleux outil qu'est Objective Caml.

Table des matières

Introduction

Pourquoi des objets ?

Les objets se sont imposés comme un concept informatique incontournable. Leur principal atout est de permettre une meilleur modularité : les classes définissent un découpage du code en différents composants ayant une interface bien définie ; l'héritage permet d'étendre et de modifier ces composants avec une grande souplesse. Comme les modules de ML remplissent une fonction similaire, le besoin d'objets se fait beaucoup moins sentir dans ML que dans d'autres langages possédant un système de modules plus faible. Le mécanisme d'héritage n'a cependant pas d'équivalent en souplesse au niveau des modules. De plus, il existent divers domaines dans lesquels les objets excellent alors que les constructions disponibles dans ML s'avèrent peu adaptés. Les objets sont alors typiquement utilisés comme des sortes de modules de première classe, que des enregistrements contenant des fonctions ne permettent de modéliser qu'imparfaitement dans ML. Un premier de ces domaines est l'écriture d'interfaces graphiques, où les divers composants graphiques se manipulent très naturellement comme des objets. Un second domaine est celui des services distribués. Par exemple, un client peut se connecter dynamiquement à différents serveurs. L'interface de ces serveurs est typiquement similaire, et peut être matérialisée par un ensemble de fonctions que l'on peut grouper sous forme d'un objet. Ces différents points justifient l'importance d'une extension de ML avec des objets.

Enjeux et difficultés

L'expérience montre qu'étendre ML avec des objets est un problème difficile. Ainsi, malgré les nombreux travaux effectués sur le sujet, Objective Caml reste toujours le seul langage fonctionnel possédant des objets et ayant dépassé le stade de prototype. La difficulté majeure réside dans la synthèse du type. Il nous paraissait essentielle de conserver cette caractéristique clé de ML, qui impose une contrainte très forte sur le système de types. Mais d'un autre côté, nous ne voulions pas que cette contrainte nuise à l'expressivité du système de types, que nous souhaitions au moins comparable à celui des langages à objets existants. Nous pensons qu'Objective Caml représente un très bon compromis entre expressivité et facilité d'utilisation, remplissant ces deux objectifs.

Notre contribution

L'étude par Didier Rémy des enregistrements extensibles et de leur typage dans ML à servi de point de départ à cette thèse. Un objet, vu de l'extérieur, se comporte en effet comme un enregistrement dont les champs sont ses méthodes. Un important travail restait à effectuer afin de mettre en œuvre ce résultat en pratique. Tout d'abord, le système de types proposé par Didier Rémy est structurelle : le type d'un enregistrement n'y est pas un constructeur opaque comme en ML, mais est défini comme la liste des types de ses champs. Par conséquent, le type d'un objet est potentiellement très grand. Il peut en effet contenir les types de nombreuses méthodes, et ces types peuvent eux-même contenir des types d'objets. Il était donc impératif de développer un mécanisme d'abréviations suffisamment puissant et facile à utiliser pour contrer cette difficulté. Par ailleurs, alors que le type des objets était pratiquement dicté par notre choix de système de types, le typage des classes restait entièrement à concevoir. Enfin, il fallait également obtenir une compilation efficace des objets.

Notre travail ne s'est pas limité à la conception d'Objective Caml, et nous avons également suivi différentes pistes à la recherche d'améliorations possibles. Ainsi, les classes d'Objective Caml peuvent avoir des méthodes privées, qui ne sont pas nécessairement visibles dans leurs sous-classes. Les méthodes publiques doivent cependant rester présentes. Cette limitation nuit à la modularité. Riecke et Stone [30] ont montré qu'il était possible de concevoir un langage dans lequel n'importe quelle méthode peut être masquer *a posteriori*. Leur résultat ne s'étend cependant pas directement à un langage avec méthodes binaires. Nous avons montré comment le faire, par une différente approche du problème. Une autre direction considérée a été le rapprochement des modules et des classes. Il serait en effet souhaitable d'éviter la présence de deux constructions aussi similaires dans Objective Caml. Nous avons ainsi proposé une fusion des deux constructions.

Plan

Le premier chapitre est un chapitre d'introduction. Des notions générales dont nous ferons usage par la suite y sont présentées. Nous commençons par décrire rapidement un langage à objets afin d'en exposer les concepts principaux. Puis nous présentons Mini-ML, un calcul reprenant les notions essentielles de ML. Finalement, nous décrivons une extension simple de ce calcul avec des objets.

Le deuxième chapitre est consacré à une formalisation de la partie objets d'Objective Caml, que nous appelons Objective ML. Afin de clarifier la présentation, nous commençons par présenter une version simplifiée de ce calcul. Puis nous décrivons la version complète et en montrons sa correction. Le calcul présenté n'a pas de classe. Nous décrivant donc finalement comment l'étendre avec des classes.

Le troisième chapitre détaille la synthèse de type. Dans une première partie, nous décrivons l'algorithme de synthèse et montrons sa correction et complétude. Le mécanisme d'abréviations de types d'Objective Caml est exposé dans la seconde partie détaille .

Le quatrième chapitre présente la mise en œuvre des résultats précédents dans Objective Caml. Il décrit la compilation des objets et des classes.

Enfin, les deux derniers chapitres explorent des possibles variantes du langage. Le cinquième chapitre décrit ainsi un calcul d'objet dans lequel le masquage des méthodes d'une classe est possible *a posteriori* même en présence de méthodes binaires. Le sixième chapitre montre comment de fusionner classes et modules en une même construction.

Chapitre 1

Généralités

Nous commençons ce chapitre en précisant quelques notations générales utilisées dans le reste de ce document. Puis nous faisons une présentation rapide d'un langage à objets, ce qui nous permet d'introduire les principaux concepts associés aux objets. Dans un troisième temps, nous décrivons un noyau pur de ML et définissons sa sémantique et son système de type. Nous donnons alors une extension simple de ce calcul avec des objets, montrant ainsi comment ceux-ci peuvent être typés dans ML. Enfin, nous discutons de ce qui manque à ce calcul pour obtenir un langage à objets suffisamment expressif.

1.1 Quelques notations

Le domaine d'une fonction f est notée $\operatorname{dom} f$. L'image de cette fonction est noté $\operatorname{im} f$. La restriction d'une fonction à un domaine A est notée $f|_A$. La fonction notée $f; (x = a)$, ou $f; (x : a)$, est la fonction qui à x associe a et qui est autrement définie comme f. Plus généralement, nous définissons $f; f'$ par

$$\begin{array}{rcl} (f; f')(x) & = & f'(x) \text{ si } x \in \operatorname{dom} f' \\ & = & f(x) \text{ sinon} \end{array}$$

Nous utilisons la notation $f \subseteq g$ pour indiquer que la fonction g étend la fonction f, c'est-à-dire, que le graphe de g contient celui de f.

Nous utilisons souvent des séquences, définies par une grammaire de la forme $L ::= \emptyset \mid L; (x = a)$. Ces séquences peuvent être considérées comme des fonctions en utilisant la définition suivante :

$$\begin{array}{rcl} (L; (x = a))(x) & = & a \\ (L; (x' = a))(x) & = & L(x) \text{ si } x \neq x' \end{array}$$

Si L est une telle séquence, on se permettra donc de la considérer comme une fonction et d'écrire, par exemple, $L(x)$, et nous noterons $L|_A$ la séquence semblable à L mais ne contenant que des liaisons de domaine A.

On définit également la concaténation de deux séquences par

$$\begin{array}{rcl} L; \emptyset & = & L \\ L; (L'; (x = a)) & = & (L; L'); (x = a) \end{array}$$

On vérifie que cette notation est cohérente avec celle similaire pour les fonctions.

1.2 Introduction aux objets

Nous faisons ici une présentation rapide d'Objective Caml, afin d'introduire les caractéristiques typiques d'un langage à objets.

Un *objet* est composé d'un ensemble de valeurs, ses *variables d'instance* et d'un ensemble de fonctions, des *méthodes* permettant de manipuler ces valeurs.

Les objets sont créés à l'aide de *classes*, qui décrivent leur contenu. Voici par exemple une classe définissant des points :

```
class point =
  object
    val mutable x = 0
    method position = x
    method déplace d = x <- x + d
  end;;
```

Les objets de cette classe ont une variable d'instance **x** et deux méthodes **position** et **déplace**. La variable **x** est modifiable (mot clé **mutable**) et sa valeur initiale est 0. La méthode **position** retourne la valeur de **x**. Enfin, la méthode **déplace** prend un argument **d** et l'ajoute à la valeur de **x**.

La construction **new point** permet de créer une *instance* de cette classe, c'est-à-dire de dériver un objet de cette classe.

Seules les méthodes sont accessibles depuis l'extérieur d'un objet. Le type d'un objet ne contient donc que la liste de ses méthodes avec leur types. Ainsi, le type d'un objet de la classe `point` est $\langle \textit{position} : \textit{int}; \textit{déplace} : \textit{int} \rightarrow \textit{unit}\rangle$.

L'expression `p#position` appelle la méthode `position` de l'objet `p`. Pour un objet nouvellement créé de la classe `point`, la valeur de cette expression serait `0`. Après l'évaluation de l'expression **p#déplace** `3`, un nouvel appel de la méthode `position` retournerait `3`.

La définition d'une méthode peut faire référence à l'objet auquel elle appartient. Une méthode peut de cette façon appeler une autre méthode du même objet. Pour cela, une variable représentant cet objet (nommée dans la classe suivante **moi**) est introduite au début du corps de la classe. La méthode **affiche** l'utilise pour appeler la méthode `position`.

```
class point_imprimable init =
  object (moi)
    val mutable x = init
    method position = x
    method déplace d = x <- x + d
    method affiche = print_int (moi#position)
  end;;
```

Il est possible de dériver une nouvelle classe à partir d'une ou plusieurs classes existantes, par *héritage*. La classe fille reprend les méthodes et les variables d'instances de ses parents et peut en modifier les définitions et rajouter de nouveaux composants. On peut ainsi partager du code entre plusieurs objets similaires. La classe `point_coloré` ci-dessous reprend ainsi la définition de la classe `point` et y ajoute la méthode `couleur`.

```
class point_coloré c =
  object
    inherit point
    method couleur = c
```

```
  end;;
```

Une classe peut contenir des méthodes qui ne sont pas encore définies, mais peuvent déjà être appelées par d'autres méthodes : ce sont des méthodes *abstraites*. Elle devront être définies dans une sous-classe pour que l'on puisse créer une instance de cette sous-classe. La classe suivante définit ainsi le schéma de la classe `point` :

```
class point_abstrait =
  object
    method virtual position : int
    method virtual déplace : int -> unit
  end;;
```

La classe qui suit fournit une définition de la méthode `déplace`, qui utilise la méthode `position`

```
class point_moins_abstrait =
  object (moi)
    inherit point_abstrait
    val mutable x = 0
    method déplace d = x <- moi#position + d
  end;;
```

Lorsque l'on hérite d'une classe, on a parfois besoin de modifier la définition d'une méthode tout en gardant un accès à la définition précédente. Dans l'exemple suivant, l'identificateur `père` permet d'accéder aux définitions des méthodes héritées de la classe parente `point_imprimable`. En particulier, la méthode `affiche` peut utiliser sa définition précédente.

```
class point_coloré_imprimable init c =
  object (moi)
    inherit point_imprimable as père
    method couleur = c
    method affiche =
      print_string "(";
      père#affiche;
      print_string ", ";
      print_string (moi#couleur);
      print_string ")"
  end;;
```

Il est possible de définir des méthodes qui ne seront pas accessibles depuis l'extérieur des objets d'une classe. Ces méthodes *privées* ne peuvent être appelées que depuis d'autres méthodes du même objet. Ainsi, le type d'un objet de la classe `point_restreint` ci-dessous est $\langle position : int; pousse : unit \rangle$.

```
class point_restreint init =
  object (moi)
    val mutable x = init
    method position = x
    method private déplace d = x <- x + d
    method pousse = moi#déplace 1
  end;;
```

Les méthodes privées sont héritées.

La construction suivante permet de définir un type de classe :

```
class type restriction =
  object
    method position : int
    method pousse : unit
  end;;
```

Ce type peut être utilisé pour masquer les méthodes privées et les variables d'instance d'une classe, à l'aide d'une contrainte de type. Ainsi, seules les méthodes publiques `position` et `pousse` sont encore visibles dans la classe ci-dessous (la méthode `déplace` n'est plus directement accessible) :

```
class point_plus_restreint = (point_restreint : restriction);;
```

Une méthode peut prendre parmi ses arguments un objet de même type que l'objet auquel elle appartient. Une telle méthode s'appelle une méthode *binaire*. Dans la classe suivante, la variable de type `'a` est liée au début du corps de la classe au type de `moi`. La méthode `égale` est donc une méthode binaire.

```
class virtual comparable =
  object (moi : 'a)
    method virtual égale : 'a -> bool
  end;;
```

On peut hériter de cette classe afin de définir des points pouvant être comparés :

```
class point_comparable =
  object
    inherit point
    inherit comparable
    method égale p = (x = p#position)
  end;;
```

Le type d'un objet de cette classe est récursif : $\mu(\alpha)\langle position : int; déplace : int \to unit; égale : \alpha \to bool\rangle$. Ce type se lit comme le type infini τ vérifiant l'égalité : $\tau = \langle position : int; déplace : int \to unit; égale : \tau \to bool\rangle$.

1.3 Mini-ML

Cette section introduit un petit calcul possédant les traits principaux de ML. Ce calcul servira de base aux différents calculs d'objets que nous décrirons par la suite. Cette présentation nous donne également l'occasion de fixer de nombreuses définitions et notations.

1.3.1 Syntaxe

Les *expressions* du calcul sont définies par la grammaire suivante :

$$
\begin{array}{rll}
a ::= & x & \text{Variable} \\
 | & \lambda(x)a & \text{Abstraction} \\
 | & a(a) & \text{Application} \\
 | & \texttt{let}\ x = a\ \texttt{in}\ a & \text{Définition}
\end{array}
$$

Dans cette grammaire, x est élément d'un ensemble infini d'*identificateurs*. L'expression $\lambda(x)a$ représente une fonction de paramètre x et de corps a. L'expression $a(a')$ dénote l'application d'une fonction a à un argument a'. L'expression $\texttt{let } x = a \texttt{ in } a'$ donne à x la valeur de l'expression a dans l'expression a'.

On définit par induction sur la syntaxe la notion de *variables libres* d'une expression :

$$\begin{array}{rcl}
\mathrm{VL}(x) & = & \{x\} \\
\mathrm{VL}(\lambda(x)a) & = & \mathrm{VL}(a) \setminus \{x\} \\
\mathrm{VL}(a(a')) & = & \mathrm{VL}(a) \cup \mathrm{VL}(a') \\
\mathrm{VL}(\texttt{let } x = a' \texttt{ in } a) & = & (\mathrm{VL}(a) \setminus x) \cup \mathrm{VL}(a')
\end{array}$$

Les variables apparaissant dans une expression et qui ne sont pas libres sont appelées *variables liées*. Deux expressions seront considérées égales si elles ne diffèrent que par un renommage de leurs variables liées.

1.3.2 Sémantique

Définir une *sémantique* consiste à donner une signification aux expressions du langage. Plusieurs types de sémantiques existent. Celle que nous allons présenter est une sémantique *opérationnelle*, décrivant comment les expressions sont évaluées. Cette sémantique est définie par des règles de *réécriture* : nous allons indiquer comment se transforment certaines portions d'expressions ainsi que les endroits dans une expression où une telle transformation peut avoir lieu. Nous définissons ainsi une étape de réduction. L'évaluation consiste en une suite de telles étapes. Elle peut ne jamais s'arrêter (elle *diverge*), ou se terminer sur une expression sur laquelle aucune règle ne s'applique. Un ensemble de *valeurs* définit les expressions sur lesquelles l'évaluation est autorisée à s'interrompre. Les autres expressions auxquelles on peut aboutir correspondent à des erreurs.

Les valeurs de ce calcul sont les fonctions :

$$v ::= \lambda(x)a$$

Il y a deux règles de réduction, indiquant comment une application de fonction et une définition se transforment. Elles définissent une *relation de réduction locale* $\longrightarrow$.

$$\begin{array}{rcl}
(\lambda(x)a)(v) & \longrightarrow & a\{v/x\} \\
\texttt{let } x = v \texttt{ in } a & \longrightarrow & a\{v/x\}
\end{array}$$

L'application d'une fonction à une valeur v consiste à remplacer le paramètre x de la fonction par cette valeur dans le corps de la fonction. De manière similaire, pour une définition, la variable x est remplacée par sa valeur v dans l'expression a (on pourrait en fait considérer l'expression $\texttt{let } x = a' \texttt{ in } a$ comme du sucre syntaxique, représentant l'expression $(\lambda(x)a)(a')$). Ce remplacement, noté $a\{v/x\}$, s'appelle une *substitution*. Les substitutions sont définies comme suit, par induction sur la syntaxe (dans le cas des fonctions et du `let`, un renommage des variables peut être nécessaire) :

$$x\{v/x\} \quad = \quad v$$

$$\begin{array}{rcll}
x'\{v/x\} & = & x' & \text{si } x' \neq x \\
(\lambda(x')a)\{v/x\} & = & \lambda(x')(a\{v/x\}) & \\
 & & & \text{si } x' \text{ n'est pas libre dans } v \text{ et } x' \neq x. \\
(a(a'))\{v/x\} & = & a\{v/x\}(a'\{v/x\}) & \\
(\texttt{let } x' = a' \texttt{ in } a)\{v/x\} & = & \texttt{let } x' = a'\{v/x\} \texttt{ in } a\{v/x\} & \\
 & & & \text{si } x' \text{ n'est pas libre dans } v \text{ et } x' \neq x.
\end{array}$$

Il est à noter qu'une application n'est réduite que lorsque son argument est une valeur (*appel par valeur*).

Les deux opérations de réduction présentées plus haut ne peuvent pas avoir lieu n'importe où dans une expression. Par exemple, pour une application $a(a')$, on décide de commencer par évaluer la fonction a avant d'évaluer son argument a'. On peut ainsi s'assurer que la réduction est déterministe. Ce contrôle est effectué en utilisant des contextes d'évaluation. Ceux-ci sont définis par la grammaire suivante :

$$\begin{array}{rl}
E ::= & [_] \\
 & \mid E(a) \\
 & \mid v(E) \\
 & \mid \texttt{let } x = E \texttt{ in } a
\end{array}$$

Le trou $[\_]$ indique où peut avoir lieu la réduction. Ces contextes permettent d'étendre la relation de réduction locale en une *relation de réduction* portant sur l'ensemble des termes et définissant une étape de réduction par la règle suivante : $E[a] \longrightarrow E[a']$ ssi $a \longrightarrow a'$.

On vérifie que si une expression a se met sous la forme $E[a']$ où a' peut être réduite localement, alors cette décomposition est unique. La relation de réduction est donc bien déterministe.

1.3.3 Typage

Nous allons maintenant définir des *règles de typage* pour le calcul. Ces règles permettent d'associer aux expressions des *types*. Une expression a est *erronée* si elle n'est pas une valeur et s'il n'existe pas d'expression a' telle que $a \longrightarrow a'$. Les règles de typage sont choisies de manière à ce qu'une expression *typable* (c'est-à-dire, à laquelle on peut donner un type) ne se réduise jamais en une expression erronée.

Type

Les types sont définis par la grammaire suivante :

$$\tau ::= \alpha \mid \tau \to \tau$$

Un type est soit une *variable de type* α, appartenant à un ensemble infini de variables, soit un *type fonctionnel* $\tau' \to \tau$, correspondant à une fonction prenant en argument une valeur de type τ' et retournant une valeur de type τ.

Substitution de type

Les substitutions permettent de formaliser la spécialisation d'un type, obtenue en remplaçant certaines de ses variables par d'autres types.

Une *substitution* θ est définie comme une application des variables de type dans les expressions de type, égale à l'identité partout sauf pour un nombre fini de variables. On l'étend de manière naturelle par induction en un automorphisme d'expressions de type, que l'on notera encore $\theta : \theta(\tau' \to \tau) = \theta(\tau') \to \theta(\tau)$.

On notera $\tau\{\tau'/\alpha\}$ l'application à τ de la substitution qui associe τ' à α (et qui laisse les autres variables de type inchangées). Plus généralement, on notera $\tau\{\vec{\tau}/\vec{\alpha}\}$ le remplacement simultané dans τ d'un ensemble de variables de types $\vec{\alpha}$ par des types correspondants $\vec{\tau}$.

Schéma de type

On veut pouvoir représenter une classe de types ayant la même structure par un seul type, par exemple pouvoir parler de l'ensemble des types ayant la forme $\tau \to \tau$ (par exemple, l'identité, $\lambda(x)x$, prend en argument une valeur de n'importe quel type τ et renvoie une valeur de même type). On utilise pour cela des *schémas de types* : ce sont des types quantifiés en tête, comme par exemple $\forall(\alpha)\alpha \to \alpha$. Les schémas de types sont décrits par la grammaire suivant :

$$\sigma ::= \forall(\vec{\alpha}_i)\tau \qquad \text{Schéma de type}$$

où $\{\vec{\alpha}_i\}$ est un ensemble de variables de type.

De même que pour les expressions, on définit les variables libres d'un schéma de type :

$$\begin{aligned} \mathrm{VL}(\alpha) &= \{\alpha\} \\ \mathrm{VL}(\tau' \to \tau) &= \mathrm{VL}(\tau) \cup \mathrm{VL}(\tau') \\ \mathrm{VL}(\forall(\vec{\alpha}_i)\tau) &= \mathrm{VL}(\tau) \setminus \{\vec{\alpha}_i\} \end{aligned}$$

Deux schémas de types ne différant que par un renommage des variables liées (α-conversion) sont identifiés.

Les substitutions sont étendues aux schémas de type : $\theta(\forall(\vec{\alpha}_i)\tau) = \forall(\vec{\alpha}_i)\theta(\tau)$, pourvu que $\theta(\alpha_i) = \alpha_i$ pour tout i et $\{\vec{\alpha}_i\} \cap \mathrm{VL}(\theta(\mathrm{VL}(\tau) \setminus \{\vec{\alpha}_i\})) = \emptyset$.

Environnement de typage

Un *environnement* Γ est une application finie des variables x du calcul vers les schémas de type. Il permet de donner un type aux variables libres d'une expression au cours du typage.

Les variables de types libres d'un environnement sont les variables libres des types qu'il contient : $\mathrm{VL}(\Gamma) = \bigcup_{x \in \mathrm{dom}\,\Gamma} \mathrm{VL}(\Gamma(x))$.

L'opérateur de *généralisation* construit un schéma de type à partir d'un type et d'un environnement de typage, comme suit :

$$\mathrm{gen}(\tau, \Gamma) = \forall(\vec{\alpha}_i)\tau \text{ avec } \{\vec{\alpha}_i\} = \mathrm{VL}(\tau) \setminus \mathrm{VL}(\Gamma)$$

L'idée est que si une variable apparaissant dans un type n'est pas fixée par l'environnement de typage, alors on peut en fait lui donner n'importe quelle valeur.

Règles de typage

Une *règle de typage* est une *règle d'inférence* :

$$\frac{P_1 \; P_2 \; ... \; P_n}{P}$$

Elle permet d'établir que la *conclusion* P est vraie à partir du moment où les *prémisses* $P_1, \ldots, P_n$ le sont.

Les règles de typage définissent un jugement de typage $\Gamma \vdash a : \tau$. Ce jugement se lit : « dans l'environnement Γ, l'expression a a le type τ ». C'est la plus petite relation entre un environnement, une expression et un type engendrée par les règles de typage ci-dessous.

La règle de typage des identificateurs indique que le type τ' d'un identificateur x est une *instance* du schéma de type $\forall(\vec{\alpha})\tau$ qui lui est associé dans l'environnement Γ, c'est-à-dire qu'il existe une substitution θ ne modifiant que les variables $\vec{\alpha}$ telle que $\tau' = \theta(\tau)$.

$$\text{(Var)} \quad \frac{(x : \forall(\vec{\alpha})\tau) \in \Gamma}{\Gamma \vdash x : \tau\{\vec{\tau}/\vec{\alpha}\}}$$

Une fonction a le type $\tau' \to \tau$ si son corps a a le type τ en faisant l'hypothèse supplémentaire que son paramètre x a le type τ'. Le type de l'application d'une fonction a de type $\tau' \to \tau$ à un argument a' est τ, pourvu que le type de l'argument soit τ'.

$$\text{(Abs)} \quad \frac{\Gamma; x : \tau' \vdash a : \tau}{\Gamma \vdash \lambda(x)a : \tau' \to \tau} \qquad \text{(App)} \quad \frac{\Gamma \vdash a : \tau' \to \tau \quad \Gamma \vdash a' : \tau'}{\Gamma \vdash a(a') : \tau}$$

Pour une définition, le typage de l'expression a' est fait en supposant que le type de la variable x est une généralisation du type τ de l'expression a. Le type de x pourra donc être spécialisé différemment par la règle Var à chaque utilisation de cette variable dans a'.

$$\text{(Let)} \quad \frac{\Gamma \vdash a : \tau \quad \Gamma; x : \mathrm{gen}(\tau, \Gamma) \vdash a' : \tau'}{\Gamma \vdash \texttt{let } x = a \texttt{ in } a' : \tau'}$$

1.3.4 Correction

On doit s'assurer de la *correction* des règles de typage vis-à-vis de la sémantique du calcul, c'est-à-dire montrer que toute expression à laquelle on peut donner un type s'exécutera « correctement ». L'objet de cette section est de formaliser cette notion de correction.

Nous commençons par quelques définitions. On note $\longrightarrow^*$ la fermeture réflexive transitive de la relation de réduction $\longrightarrow$. La réduction d'une expression a peut soit se terminer sur une expression *finale* a' ($a \longrightarrow^* a'$ et il n'existe pas d'expression a'' telle que $a' \longrightarrow a''$), soit diverger (ce que l'on notera $a\Uparrow$). On dit d'autre part qu'une expression est *bien typée* si l'on peut lui assigner un type dans l'environnement vide ($\vdash a : \tau$).

Un système de type est correct vis à vis d'une sémantique donnée si les expressions bien typées divergent ou s'évaluent en une valeur :

> *Si $\vdash a : \tau$, alors soit $a\Uparrow$, soit il existe une valeur v telle que $a \longrightarrow^* v$.*

En fait, on peut généralement caractériser plus finement la valeur obtenue après réduction :

> *Si $\vdash a : \tau$, alors soit $a\Uparrow$, soit il existe une valeur v telle que $a \longrightarrow^* v$ et $\vdash v : \tau$.*

La preuve de cette dernière propriété se fait en général en deux étapes. On montre d'abord que la réduction préserve les types :

> *Si $\Gamma \vdash a : \tau$ et $\Gamma \vdash a \longrightarrow a'$ alors $\vdash a' : \tau$.*

Puis l'on montre que si une expression n'est pas une valeur et est bien typée, alors elle peut être réduite :

> *Si $\vdash a : \tau$ et a n'est pas une valeur, alors il existe une expression a' telle que $a \longrightarrow a'$*[1].

Il est alors facile de conclure. On peut ainsi montrer la correction du système de type que nous venons de présenter.

1.3.5 Synthèse de type

Nous allons présenter un algorithme de *synthèse de type*. Cet algorithme calcule lorsque c'est possible un type pour une expression et échoue dans le cas contraire. Cet algorithme procède par induction sur la syntaxe de l'expression et repose sur la stabilité par instanciation des jugements de type : si $\Gamma \vdash a : \tau$, alors pour toute substitution θ, $\theta(\Gamma) \vdash a : \theta(\tau)$. En effet, dans le cas d'une application par exemple, si on a montré que $\Gamma \vdash a : \tau' \rightarrow \tau$ et que $\Gamma \vdash a' : \tau_1$, il suffit de trouver une substitution θ telle que $\theta(\tau') = \theta(\tau_1)$ pour en déduire $\theta(\Gamma) \vdash a(a') : \theta(\tau)$.

La recherche d'une telle substitution s'appelle un problème d'*unification*. Si un problème d'unification a une solution, on montre l'existence d'une solution *principale* θ_0, plus générale que toute autre solution : pour toute substitution θ vérifiant $\theta(\tau) = \theta(\tau')$, il existe une substitution θ' telle que $\theta = \theta' \circ \theta_0$. Une solution principale indique les modifications minimales devant être apportées aux types τ et τ' pour les égaler. Elle est unique à renommage près des variables de son image. L'algorithme de Robinson [31] permet de calculer une telle solution, que nous noterons $\mathrm{mgu}(\tau, \tau')$. Cette solution a la propriété supplémentaire de ne pas introduire de nouvelles variables de type.

Un algorithme de synthèse de type est présenté en figure 1.1. C'est une variante de l'algorithme de Damas et Milner [9]. Cet algorithme prend en entrée une expression a, un environnement Γ et un ensemble V de variables de types « fraîches ». En cas de succès, il retourne un type τ et une substitution θ tels que

[1] Les seules valeurs de Mini-ML étant les fonctions, la preuve de cette propriété s'en trouve légèrement simplifiée. Les preuves de correction sont donc généralement établies pour des calculs possédant également des constantes. Nous avons choisi de ne pas avoir de constantes car nous pensons que celles-ci encombreraient inutilement les calculs que nous verrons par la suite. Ces calculs auront de toute manière une autre sorte de valeur : des objets.

Infère(Γ, a, V) est le triplet (τ, θ, V') défini par :

- Si $a = x$:
  Alors $(\tau, V') = \text{Instance}(\Gamma(x), V)$ et θ est l'identité.
- Si $a = \lambda(x)a_1$:
  Soit $\alpha \in V$.
  Soit $(\tau_1, \theta_1, V_1) = \text{Infère}((\Gamma; x : \alpha), a_1, V \setminus \{\alpha\})$.
  Alors $\tau = \theta_1(\alpha) \to \tau_1$, $V' = V_1$ et $\theta = \theta_1$.
- Si $a = a_1(a'_1)$:
  Soit $(\tau_1, \theta_1, V_1) = \text{Infère}(\Gamma, a_1, V)$.
  Soit $(\tau_2, \theta_2, V_2) = \text{Infère}(\theta_1(\Gamma), a_2, V_1)$.
  Soit $\alpha \in V_2$ et $\theta_3 = \text{mgu}(\theta_2(\tau_1), \tau_2 \to \alpha)$
  Alors $\tau = \theta_3(\alpha)$, $V' = V_2 \setminus \{\alpha\}$ et $\theta = \theta_3 \circ \theta_2 \circ \theta_1$.
- Si $a = \texttt{let}\ x = a'_1\ \texttt{in}\ a_1$:
  Soit $(\tau_1, \theta_1, V_1) = \text{Infère}(\Gamma, a'_1, V)$.
  Soit $(\tau_2, \theta_2, V_2) = \text{Infère}((\theta_1(\Gamma); x : \text{gen}(\tau_1, \theta_1(\Gamma))), a_1, V_1)$.
  Alors $\tau = \tau_2$, $V' = V_2$ et $\theta = \theta_2 \circ \theta_1$.

FIG. 1.1: Algorithme de synthèse de type

$\theta(\Gamma) \vdash a : \tau$, ainsi qu'un sous-ensemble V' de V. Il utilise une fonction Instance qui retourne une instance triviale d'un schéma de type σ : cette fonction vérifie les conditions suivantes :

$$\begin{aligned} \text{Instance}(\tau, V) &= (\tau, V) \\ \text{Instance}(\forall(\alpha)\sigma, V) &= (\sigma\{\alpha'/\alpha\}, V \setminus \{\alpha'\}) \text{ où } \alpha' \in V \end{aligned}$$

On montre que cet algorithme est correct (s'il réussit, alors on a bien $\theta(\Gamma) \vdash a : \tau$) et complet (s'il existe une substitution θ' et un type τ' tels que $\theta'(\Gamma) \vdash a : \tau'$, alors l'algorithme réussit). On montre de plus qu'il calcule un type τ et une substitution θ qui sont les plus généraux possibles : pour tout jugement $\theta'(\Gamma) \vdash a : \tau'$, il existe une substitution $\theta' = \theta'' \circ \theta$ et un type $\tau' = \theta''(\tau)$. On dit que le type τ est le *type principal* de a.

1.4 Un petit calcul d'objets

Nous allons maintenant présenter une extension de Mini-ML avec des objets. Cette extension est très simple, mais permet de donner une idée de la manière dont les objets peuvent être typés.

1.4.1 Syntaxe

Nous définissons la syntaxe de ce calcul en étendant la syntaxe de Mini-ML (section 1.3.1 page 12). La syntaxe des objets est décrite ci-dessous. Cette grammaire utilise un ensemble infini de noms de méthodes notés par la lettre l. Un objet $[L]$ contient une collection L de méthodes, qui associe à chaque méthode l de l'objet sa définition $\varsigma(x)a$. Cette définition consiste en une expression

a, le corps de la méthode, paramétrée par une variable x représentant l'objet lui-même. L'appel de méthode se fait à l'aide de la notation $a\#l$.

$$
\begin{array}{rll}
a ::= & \ldots & \\
& \mid [L] & \text{Objet} \\
& \mid a\#l & \text{Appel de méthode} \\
L ::= & \emptyset & \text{Objet vide} \\
& \mid L; (l = \varsigma(x)a) & \text{Méthode}
\end{array}
$$

La grammaire des types est étendue avec des types objets et des types récursifs.

$$
\begin{array}{rll}
\tau ::= & \ldots \mid \langle\omega\rangle \mid \mu(\alpha)\tau & \text{Type} \\
\omega ::= & \rho \mid \emptyset \mid l : \tau; \omega &
\end{array}
$$

Le type d'un objet $\langle\omega\rangle$ est constitué d'une *rangée* ω associant à chaque méthode l de l'objet son type τ. Cette rangée se termine soit par une *variable de rangée* ρ, soit par la rangée vide $\emptyset$. Dans le premier cas, seule une partie du type est spécifiée et le type pourra être raffiné plus tard par instanciation. Par exemple, le type $\langle l : \tau; \rho\rangle$ est le type d'un objet ayant une méthode l de type τ, mais pouvant avoir également d'autres méthodes, représentées par la variable ρ. Une même méthode ne doit pas apparaître deux fois dans le type d'un objet. Cela peut être aisément assuré en classant les types suivant des sortes [24, 25]. La distinction entre les types τ et ω (et entre les variables de types α et ρ) peut être également assurée par des sortes. On les confondra donc par la suite lorsqu'il n'y aura pas d'ambiguïté.

Un type récursif $\mu(\alpha)\tau[\alpha]$ est moralement le point fixe de la fonction qui à un type τ_0 associe le type $\tau[\tau_0]$. Les types récursifs permettent par exemple d'exprimer qu'une méthode d'un objet retourne un objet du même type :

$$\mu(\alpha)\langle clone : \alpha\rangle$$

L'ordre des méthodes n'ayant pas d'importance, l'égalité sur les types est définie modulo la famille suivante d'axiomes de commutation à gauche des champs du type d'un objet :

$$(l_1 : \tau_1; l_2 : \tau_2; \tau) = (l_2 : \tau_2; l_1 : \tau_1; \tau)$$

L'égalité entre types récursifs est quant à elle définie par les règles habituelles [3] suivantes :

$$
\text{(Rec)}\quad \frac{\tau_1 = \tau_2}{\mu(\alpha)\tau_1 = \mu(\alpha)\tau_2}
\qquad
\text{(Déroule-Enroule)}\quad \mu(\alpha)\tau = \tau\{\mu(\alpha).\tau/\alpha\}
$$

$$
\text{(Contraction)}\quad \frac{\tau_1 = \tau\{\tau_1/\alpha\} \wedge \tau_2 = \tau\{\tau_2/\alpha\} \qquad \mu(\alpha)\tau \text{ bien formé}}{\tau_1 = \tau_2}
$$

La règle Rec indique que deux types récursifs sont égaux s'ils ont la même structure. La règle Déroule-Enroule indique qu'un type récursif peut être déroulé ou enroulé librement. Enfin, la règle Contraction permet de vérifier l'égalité de deux types récursifs même s'ils ont une « période » différente (par exemple, elle permet de montrer que les types $\mu(\alpha)\tau[\alpha]$ et $\mu(\alpha)\tau[\tau[\alpha]]$ sont égaux.

On dira qu'un type récursif $\mu(\alpha)\tau$ est *bien formé* si le type τ n'est ni une variable, ni de la forme $\mu(\alpha')\tau'$ (ce n'est pas trop restrictif car $\mu(\alpha)\mu(\alpha')\tau'$ peut toujours être réécrit en $\mu(\alpha)(\tau'\{\alpha/\alpha'\})$. Cette règle de bonne formation assure que les types récursifs puissent être vus comme des arbres rationnels. On ne considérera par la suite que des types bien formés.

1.4.2 Sémantique

Nous décrivons maintenant les extensions apportées à la sémantique précédente (section 1.3.2 page 13). Celle-ci doit être modifiée afin de prendre en compte les appels de méthodes. Les objets sont aussi des valeurs :

$$v ::= \ldots \mid [L]$$

Les contextes d'évaluation autorisent le calcul de la valeur d'un objet avant un appel de méthode :

$$E ::= \ldots \mid E\#l$$

Enfin, une nouvelle règle de réduction définit l'appel de méthode, qui consiste à extraire le corps de la méthode a de l'objet et à y remplacer la variable x par l'objet lui-même.

$$[L]\#l \quad \longrightarrow \quad a\{[L]/x\} \qquad\qquad \text{si } L(l) = \varsigma(x)a$$

1.4.3 Typage

Les règles de typage nécessitent un nouveau jugement $\Gamma \vdash L : \mathcal{L}$ permettant de définir le type du corps L d'un objet. Dans ce jugement, $\mathcal{L}$ est le type des méthodes de l'objet. C'est une liste bornée définie comme une sous-grammaire de ω :

$$\mathcal{L} ::= \emptyset \mid l : \tau; \mathcal{L}$$

Le type $\langle\mathcal{L}\rangle$ d'un objet $[L]$ est obtenu à partir du type $\mathcal{L}$ de son corps L (règle Objet). Toutes les méthodes apparaissant dans le type de l'objet doivent bien sûr être définies pour que l'objet soit bien typé. Le corps d'un objet est typé récursivement. Partant d'une liste vide de méthodes (règle Objet-Vide), on ajoute les méthodes les unes après les autres (règle Objet-Méthode). La dernière règle spécifie le typage d'un appel de méthode : si un objet a a une méthode l de type τ, alors le résultat de l'invocation de cette méthode a le type τ.

$$\frac{\Gamma \vdash L : \mathcal{L} \quad \mathrm{dom}\,\mathcal{L} = \mathrm{dom}\,L}{\Gamma \vdash [L] : \langle \mathcal{L} \rangle} \text{ (Objet)} \qquad \frac{}{\Gamma \vdash \emptyset : \mathcal{L}} \text{ (Objet-Vide)}$$

$$\frac{\Gamma \vdash L : \mathcal{L} \quad \Gamma; x : \langle \mathcal{L} \rangle \vdash a : \mathcal{L}(l)}{\Gamma \vdash L; (l = \varsigma(x)a) : \mathcal{L}} \text{ (Objet-Méthode)} \qquad \frac{\Gamma \vdash a : \langle l : \tau; \omega \rangle}{\Gamma \vdash a\#l : \tau} \text{ (Sélection)}$$

Voyons quelques exemples. L'objet *point* suivant a une méthode x qui retourne la valeur 1, et une méthode *get_x* qui appelle la méthode x :

$$\mathit{point} ::= [x = \varsigma(s)\mathit{1}; \mathit{get\_x} = \varsigma(s)s\#x]$$

Les deux méthodes de l'objet ont donc le type *int* :

$$\mathit{point} : \langle x : \mathit{int}; \mathit{get\_x} : \mathit{int} \rangle$$

On peut également définir un objet qui compare la valeur d'une de ses méthodes à la valeur de la même méthode d'un autre objet.

$$\mathit{point\_comparable} ::= [x = \varsigma(s)x; \mathit{compare} = \varsigma(s)\lambda(y)(s\#x = y\#x)]$$

Le type le plus général que l'on puisse donner à cet objet est le suivant :

$$\forall(\rho)\langle x : \mathit{int}; \mathit{compare} : \langle x : \mathit{int}; \rho \rangle \to \mathit{bool} \rangle$$

En effet, la seule contrainte que l'on ait sur le type de l'argument de la méthode *compare* est qu'il ait une méthode x de type *int*.

On s'attend à pouvoir comparer cet objet avec lui-même :

$$\mathit{point\_comparable}\#\mathit{compare}(\mathit{point\_comparable})$$

Afin de typer cette expression, il faut pouvoir unifier le type de cet objet avec le type de l'argument de sa méthode *compare*. Le résultat de cette unification est le type récursif suivant :

$$\mu(\alpha)\langle x : \mathit{int}; \mathit{compare} : \alpha \to \mathit{bool} \rangle$$

Les types récursifs apparaissent donc naturellement dans le typage des objets.

1.4.4 Synthèse de type

Algorithme d'unification

Nous présentons un algorithme d'unification traitant simultanément les types récursifs et les types d'objets. Cet algorithme est une simplification de celui utilisé pour le typage des enregistrements dans ML-ART [25], qui est détaillé dans [26]. Le formalisme utilisé a été introduit dans [17] et [24].

Un problème d'unification est également appelé un *unificande*. C'est un multi-ensemble de multi-équations, précédé d'une liste de variables quantifiées existentiellement. On le note $\exists\, \alpha_1, \ldots \alpha_p.\, e_1 \wedge \ldots e_q$. Une *multi-équation* e est un multi-ensemble de types écrit $\tau_1 \doteq \ldots \tau_n$. L'algorithme suppose que les types récursifs $\mu(\alpha)\tau$ ont été encodés à l'aide d'équations $\exists\, \alpha.\, \alpha \doteq \tau$.

(Fusionne)
$$\frac{\alpha \doteq e \wedge \alpha \doteq e'}{\alpha \doteq e \doteq e'}$$

(Décompose)
$$\frac{f({\alpha_i}^{i\in I}) \doteq f({\alpha'_i}^{i\in I}) \doteq e}{f({\alpha_i}^{i\in I}) \doteq e \wedge (\alpha_i \doteq \alpha'_i)^{i\in I}}$$

(Généralise)
$$\frac{e[\alpha \leftarrow \tau] \qquad \alpha \notin \mathrm{VL}(\tau)}{\exists\, \alpha.\, e \wedge \alpha \doteq \tau}$$

(Commute)
$$\frac{(m_1 : \alpha_1; \alpha'_1) \doteq (m_2 : \alpha_2; \alpha'_2) \doteq e}{\exists\, \alpha'.\, (m_2 : \alpha_2; \alpha'_2) \doteq e \wedge \alpha'_1 \doteq (m_2 : \alpha_2; \alpha') \wedge \alpha'_2 \doteq (m_1 : \alpha_1; \alpha')}$$

(1) Dans la règle Décompose, f est n'importe quel symbole de type, y compris $\_ \to \_$ et $(m : \_; \_)$.
(2) Afin d'assurer la terminaison, la règle Généralise doit être restreinte au cas où τ n'est pas une variable de type et α apparaît dans la multi-équation e mais aucun des termes de e n'est α.

Fig. 1.2: Règles d'unification

Une substitution est une solution d'une multi-équation si elle rend tous ses types égaux. Une solution d'un unificande est la restriction hors des variables quantifiées existentiellement d'une solution commune à toutes ses multi-équations.

Une unificande peut être simplifiée à l'aide des règles données dans la figure 1.2. Nous avons omis les règles structurelles : associativité et commutativité de $\wedge$ et de $\doteq$, extrusion et renommage des variables liées existentiellement. Les règles Fusionne, Décompose et Généralise sont standards. La règle Fusionne réunit deux multi-équations ayant une variable en commun. La règle Décompose décompose les termes d'une multi-équation en des termes plus petits. Enfin, la règle Commute permet de simplifier une multi-équation par commutation à gauche.

L'algorithme opère par réécriture d'unificandes suivant ces règles. On montre que ce processus termine toujours et que cette simplification conserve l'ensemble des solutions d'un unificande. On met ainsi un unificande sous une forme canonique.

Un unificande sous forme canonique est *résolu* si chacune de ses multi-équations est complètement décomposée (c'est-à-dire si elle contient au plus un terme qui n'est pas une variable). On lit alors immédiatement une solution principale de cet unificande. Un unificande sous forme canonique qui n'est pas résolu n'a pas de solution : il contient deux types incompatibles (qui ne peuvent pas être identifiés).

Montrons par exemple comment se déroule l'unification des types $\langle m_1 : \alpha_1; \alpha'_1\rangle$ et $\langle m_2 : \alpha_2; \alpha'_2\rangle$. On part de l'unificande $\langle m_1 : \alpha_1; \alpha'_1\rangle \doteq \langle m_2 : \alpha_2; \alpha'_2\rangle$. On applique d'abord la règle Décompose : $(m_1 : \alpha_1; \alpha'_1) \doteq (m_2 : \alpha_2; \alpha'_2)$. La règle Commute donne alors : $\exists\, \alpha.\, (m_1 : \alpha_1; \alpha) \doteq \alpha'_2 \wedge (m_2 : \alpha_2; \alpha) \doteq \alpha'_1$. Cet unificande est résolu. Une solution principale est donc la substitution qui associe $(m_1 : \alpha_1; \alpha)$ à α'_2 et $(m_2 : \alpha_2; \alpha)$ à α'_1.

Synthèse de type

Au contraire de l'algorithme d'unification de Robinson, l'algorithme ci-dessus peut introduire des variables de type qui n'étaient pas présentes dans les types initiaux, comme le montre l'exemple donné plus haut. La fonction d'unification « mgu » doit donc prendre en paramètre en plus des deux types à unifier un ensemble V de variables fraîches. Elle retourne une paire (θ', V') composée de la substitution calculée θ' et d'un nouvel ensemble V' de variables fraîches. L'algorithme de synthèse de type doit être modifié en conséquent.

L'adaptation de cet algorithme au typage des objets est aisé. En particulier, un appel de méthode $\_\#l$ peut être simplement considéré comme une fonction de type $\forall(\alpha)\forall(\rho)\langle l : \alpha; \rho\rangle \to \alpha$ (c'est ce qui a motivé initialement l'introduction des variables de rangée pour le typage des objets).

1.4.5 Sous-typage

Supposons que l'on ait un objet « point coloré » de type $\langle x : \mathit{int}; \mathit{couleur} : \mathit{string}\rangle$. Tant que l'on n'utilise pas sa méthode *couleur*, il se comporte exactement comme un objet « point » de type $\langle x : \mathit{int}\rangle$. Il parait donc correct de pouvoir lui donner ce second type. L'intérêt est double. On peut ainsi interdire l'accès à la méthode *couleur* de cet objet. De plus, il est alors possible de manipuler de manière homogène les « points » et les « points colorés » (par exemple, les mettre dans une même liste), comme on peut leur donner un type commun.

On définit donc une relation de sous-typage sur les objets, permettant de masquer des méthodes. Cette relation s'étend de manière standard à tous les types : elle est définie comme la plus grand relation symétrique et transitive cohérente avec les règles suivantes [5, 3] :

$$\frac{\tau_1' \preceq \tau_1 \qquad \tau_2 \preceq \tau_2'}{\tau_1 \to \tau_2 \preceq \tau_1' \to \tau_2'} \qquad \frac{}{\tau \preceq \tau} \qquad \frac{\omega \preceq \omega'}{\langle\omega\rangle \preceq \langle\omega'\rangle} \qquad \frac{\tau \preceq \tau' \qquad \omega \preceq \omega'}{(l : \tau; \omega) \preceq (l : \tau'; \omega')} \qquad \frac{}{\omega \preceq \emptyset}$$

Cette relation de sous-typage est stable par instanciation : si $\tau \preceq \tau'$, alors $\theta(\tau) \preceq \theta(\tau')$ pour toute substitution θ.

On montre la correction de la règle de sous-typage implicite suivante :

$$\text{(Sous-typage)}\quad \frac{\Gamma \vdash a : \tau \qquad \tau \preceq \tau'}{\Gamma \vdash a : \tau'}$$

De plus, cette règle est suffisamment proche des autres règles de typage pour que cet ajout ne complique pas de manière significative une preuve de correction du langage.

Cependant, la synthèse de type telle que nous l'avons présentée précédemment n'est pas possible avec cette règle. En effet, l'unification ne permet pas de traiter des contraintes d'inégalité sur les types : on ne peut résoudre ainsi que des contraintes d'égalité. Il serait possible de faire de la synthèse de type à l'aide d'un algorithme basé sur la résolution de contraintes de sous-typage plutôt que sur l'unification [12, 23]. Mais cette technique n'est pas encore suffisamment maîtrisée pour que nous puissions l'utiliser dans un vrai langage. Elle produit notamment des types difficilement lisibles [11].

On est donc amené à rendre le sous-typage explicite en ajoutant une construction au calcul :

$$a ::= \ldots \mid (a : \tau :> \tau') \qquad \text{Contrainte de sous-typage}$$

Cette construction permet de considérer une expression de type τ comme étant de type τ', pourvu que ces deux types soient en relation de sous-typage :

$$\text{(CSTR)}\quad \frac{\Gamma \vdash a : \theta(\tau) \qquad \tau \preceq \tau'}{\Gamma \vdash (a : \tau :> \tau') : \theta(\tau')}\text{(pour toute substitution } \theta)$$

Du point de vue de la synthèse de type, cela correspond à introduire une famille de constantes $(\_ : \tau :> \tau')$ de type $\forall(\vec{\alpha})\tau \to \tau'$ (où $\vec{\alpha}$ sont les variables libres de τ et τ') indexées par l'ensemble des paires de types (τ, τ') telles que $\tau \preceq \tau'$.

On a alors deux options pour prouver la correction du calcul. Une première possibilité est d'ajouter des règles de réduction permettant de déplacer les contraintes de sous-typage afin qu'elles ne bloquent pas la réduction (par exemple, $(\lambda(x)a : \tau_1' \to \tau_1 :> \tau_2' \to \tau_2)(a)$ se réduirait en $\lambda(x)(a : \tau_1 :> \tau_2)((a : \tau_2' :> \tau_1'))$). L'autre solution, plus simple, consiste à prouver la correction du langage en présence de la règle SOUS-TYPAGE. La réduction peut alors tout simplement effacer les contraintes :

$$(a : \tau :> \tau') \quad \longrightarrow \quad a$$

On peut remarquer que la règle CSTR vérifie localement que les types τ et τ' sont en relation de sous-typage. Elle ne permet donc pas de typer autant d'expressions qu'à l'aide d'un système de type permettant de faire des hypothèses de sous-typage [12, 23, 7, 22, 1, 2]. Cependant, il apparaît que le polymorphisme fourni par les variables de rangées permet de se passer dans de nombreux cas de sous-typage. Par exemple, le type $\forall(\alpha \preceq \langle l : \tau\rangle)(\alpha \to \alpha)$, utilisant une quantification bornée, correspond approximativement au type $\forall(\rho)\langle l : \tau; \rho\rangle \to \langle l : \tau; \rho\rangle$. Ainsi, le fait que le sous-typage soit explicite et résolu localement est rarement une limitation.

1.5 Extensions

Il manque au calcul que nous venons de présenter de nombreuses caractéristiques que l'on aimerait trouver dans un calcul d'objets. Ainsi, la plupart des exemples donnés en section 1.2 (page 1.2) ne peuvent pas être encodés dans ce calcul. Nous allons maintenant faire un tour d'horizon des principales caractéristiques que l'on souhaite avoir et discuter des difficultés associées à leur incorporation dans un calcul d'objets.

1.5.1 Changement de la définition des méthodes

Il est important qu'un objet puisse avoir un état modifiable, ou du moins, dans un calcul sans effets de bord, que l'on puisse faire la copie d'un objet tout en modifiant une partie de son état. Par exemple, un objet « point » aurait une position et sa méthode *déplace* permettrait de changer cette position. En supposant l'existence d'une construction $a\#l \Leftarrow \varsigma(x)a'$ permettant de remplacer la définition de la méthode l de l'objet a par $\varsigma(x)a'$, un tel objet pourrait s'écrire :

$$[\mathit{position} = \varsigma(s)7;\, \mathit{d\acute{e}place} = \varsigma(s)\lambda(y)s\#\mathit{position} \Leftarrow \varsigma(s')y]$$

L'utilisation d'une méthode constante comme dans cet exemple, permet de modéliser une variable d'instance.

La sémantique d'une telle opération ne pose pas de difficulté :

$$[L]\#l \Leftarrow \varsigma(x)a \quad \longrightarrow \quad [L; (l = \varsigma(x)a)] \qquad\qquad \text{si } l \in \operatorname{dom} L$$

La règle de typage « naturelle » suivante n'est cependant pas correcte en présence de sous-typage.

$$\frac{\Gamma \vdash a : \tau \qquad \Gamma; x : \tau \vdash a' : \tau' \qquad \tau = \langle l : \tau'; \tau'' \rangle}{\Gamma \vdash a\#l \Leftarrow \varsigma(x)a' : \tau}$$

Considérons en effet un objet ayant une méthode l de type τ. D'autres méthodes de l'objet peuvent faire appel à cette méthode et font l'hypothèse qu'elle a ce type. Les règles de sous-typage nous permettent de donner à l'objet un type dans lequel la méthode l a le type τ' plus général que τ. Mais alors la règle de typage nous autorise à redéfinir la méthode de manière à ce qu'elle retourne une valeur de ce type τ', alors que les autres méthodes s'attendent à un type plus précis.

Un remède possible, mais contraignant, est d'interdire le *sous-typage en profondeur*, c'est-à-dire interdire de modifier le type des méthodes d'un objet par sous-typage. On peut relâcher cette contrainte en distinguant deux types d'objet : un type d'objet sans sous-typage en profondeur mais permettant de redéfinir les méthodes de l'objet, et un type d'objet avec sous-typage en profondeur mais n'autorisant pas les modifications de méthodes. Le passage du premier type au second s'effectue par sous-typage. C'est cette solution que nous adopterons dans le chapitre suivant.

1.5.2 Ajout de méthodes

Afin de modéliser l'héritage, on doit pouvoir ajouter à un objet de nouvelles méthodes : une classe peut alors être représentée par une fonction retournant un objet ; une sous-classe est une fonction appelant la fonction précédente et modifiant l'objet ainsi obtenu en redéfinissant certaines méthodes et en en ajoutant de nouvelles.

La règle de réduction que nous donnions précédemment permet d'ajouter une méthode l, si l'on enlève la restriction que l soit déjà définie :

$$[L]\#l \Leftarrow \varsigma(x)a \quad \longrightarrow \quad [L; (l = \varsigma(x)a)]$$

Il est cependant difficile de donner une règle de typage pour cette construction. Il faut en effet pouvoir exprimer que l'objet que l'on étend n'a initialement pas de méthode l (ou éventuellement que s'il en a déjà une, elle a un type compatible). Le système de type que nous utilisons ne permet pas d'exprimer cette contrainte. Il pourrait être enrichi dans ce but, en associant une annotation de présence à chaque méthode [25, 24]. La règle de typage serait alors :

$$\frac{\Gamma \vdash a : \langle l : \mathrm{Absent}; \tau'' \rangle \qquad \Gamma; x : \tau \vdash a' : \tau' \qquad \tau = \langle l : \mathrm{Présent}(\tau'); \tau'' \rangle}{\Gamma \vdash a\#l \Leftarrow \varsigma(x)a' : \tau}$$

Cette règle indique que l'objet résultant a une méthode l de type τ' et que l'objet d'origine ne doit pas avoir de méthode l.

Une autre solution, qui évite de compliquer le système de type, est d'exiger que la liste des méthodes déjà présentes dans l'objet soit connue, c'est-à-dire, que le type de l'objet ne soit pas extensible :

$$\frac{\Gamma \vdash a : \langle \mathcal{L} \rangle \qquad \Gamma; x : \tau \vdash a' : \tau' \qquad \tau = \langle l : \tau'; \mathcal{L} \rangle \qquad l \notin \mathrm{dom}\,\mathcal{L}}{\Gamma \vdash a\#l \Leftarrow \varsigma(x)a' : \tau}$$

Mais une telle règle pose des problèmes pour la synthèse de type, car le typage n'est plus principal : par exemple, on peut donner à une expression

$$\lambda(x)a\#l \Leftarrow \varsigma(x)a'$$

un type aussi bien de la forme $\langle l : \tau \rangle$ que de la forme $\langle l : \tau; l' : \tau' \rangle$, mais aucun type plus général que ces deux types ne convient. Il faudrait en fait que la liste des méthodes de l'objet n'ait pas besoin d'être devinée. C'est le cas si les objets extensibles ne sont pas passés en paramètre de fonction, et ne sont pas soumis au sous-typage. Bien sûr, on ne peut pas appliquer ces contraintes à tous les objets. On fera donc une distinction entre deux sortes d'objets : les *prototypes*, qui peuvent être étendus, et les objets proprement dit, qui ne peuvent l'être. Comme les prototypes ne peuvent être passés en paramètre à une fonction, ils ne sont pas des valeurs de première classe. On peut remarquer que c'est justement en général le cas des classes, qu'ils doivent permettre de modéliser.

Il existerait une autre raison pour ne pas appliquer de règle de sous-typage aux objets extensibles : la sémantique que nous venons de donner n'est en effet pas compatible avec le sous-typage. Supposons en effet que nous ayons défini un objet ayant une méthode m de type τ et une méthode n pouvant appeler cette première méthode. Le sous-typage permettrait de cacher la méthode m. On pourrait alors ajouter à l'objet une méthode de même nom, mais d'un type τ' incompatible avec τ. Mais alors la méthode n obtiendrait une valeur de type τ' par l'appel de la méthode m, alors qu'elle attend une valeur de type τ. Le sous-typage ne doit donc pas être autorisé sur les prototypes.

Il est en fait possible de conserver le sous-typage en adoptant une sémantique telle que l'ajout d'une méthode l n'interfère pas avec une méthode de même nom l déjà existante : les méthodes qui faisaient appel à l'ancienne méthode l continueront à l'utiliser. On associe pour cela à chaque objet un dictionnaire, traduisant un nom externe de méthode en un nom interne. Lorsqu'une méthode est ajoutée, un nouveau nom interne lui est associé et le dictionnaire de l'objet est modifié en conséquence. Les méthodes déjà présentes dans l'objet utilisent leur propre dictionnaire et ne sont donc pas affectées.

Chapitre 2

Objective ML

Nous présentons dans ce chapitre Objective ML, une formalisation de la partie objet du langage Objective Caml [19]. Cette formalisation est complèment différente de celle donnée dans [27]. Nous pensons que cette nouvelle formalisation est plus simple. De plus, elle permet de décrire les méthodes privées de Objective Caml qui n'étaient pas prises en compte dans [27].

Afin de rendre la présentation plus claire, nous décrivons dans un premier temps une version simplifiée du calcul servant de base à Objective ML, où les méthodes privées sont systématiquement héritées. Puis nous donnons le calcul complet et prouvons sa correction. Finalement, nous ajoutons des classes à ce calcul.

2.1 Version simplifiée du calcul

Ce calcul est basé sur Mini-ML (section 1.3 page 1.3). La présentation de ce premier calcul est divisée en trois phases. Nous décrivons d'abord les objets, puis les prototypes. Finalement, nous donnons quelques constructions qui ne sont pas essentielles mais permettent d'augmenter notablement l'expressivité du calcul.

2.1.1 Objets

Syntaxe

La syntaxe de ce calcul est donnée ci-dessous. C'est une extension de celle de Mini-ML, très similaire à celle du calcul précédent présentée dans la section 1.4.1 page 18. Les objets $[\mathcal{L}]_{\mathcal{P}}$ portent maintenant la liste $\mathcal{P}$ de leurs méthodes publiques ($\mathcal{P}$ est un ensemble de noms de méthodes). L'expression $a\#l \Leftarrow \varsigma(x)a'$ dénote le remplacement de la définition de la méthode l de l'objet a par la méthode $\varsigma(x)a'$.

$$
\begin{array}{lll}
a ::= x & & \text{Variable} \\
\quad \mid \lambda(x)a & & \text{Abstraction} \\
\quad \mid a(a) & & \text{Application} \\
\quad \mid \texttt{let}\ x = a\ \texttt{in}\ a & & \text{Définition} \\
\quad \mid [L]_{\mathcal{P}} & & \text{Objet} \\
\quad \mid a\#l & & \text{Appel de méthode} \\
\quad \mid a\#l \Leftarrow \varsigma(x)a & & \text{Redéfinition de méthode} \\
L ::= \emptyset & & \text{Objet vide} \\
\quad \mid L; (l = \varsigma(x)a) & & \text{Méthode}
\end{array}
$$

Sémantique

La sémantique de ce calcul est une extension de la sémantique de Mini-ML (donnée en section 1.3.2 page 13). Les valeurs sont les fonctions et les objets :

$$v ::= \lambda(x)a \mid [L]_{\mathcal{P}}$$

Un objet est évalué avant l'invocation ou la modification de l'une de ses méthodes. Deux cas sont donc ajoutés dans la définition des contextes :

$$
\begin{array}{l}
E ::= [\_] \\
\quad \mid E(a) \\
\quad \mid v(E) \\
\quad \mid \texttt{let}\ x = E\ \texttt{in}\ a \\
\quad \mid E\#l \\
\quad \mid E\#l \Leftarrow \varsigma(x)a
\end{array}
$$

L'appel de méthode s'effectue comme en section 1.4.2 (page 20). La modification d'une méthode consiste à insérer la nouvelle définition de cette méthode dans l'objet. Les autres règles de réduction sont inchangées.

$$
\begin{array}{rcll}
[L]_{\mathcal{P}}\#l & \longrightarrow & a\{[L]_{\mathcal{P}}/x\} & \text{si } L(l) = \varsigma(x)a \\
[L]_{\mathcal{P}}\#l \Leftarrow \varsigma(x)a & \longrightarrow & [L; (l = \varsigma(x)a)]_{\mathcal{P}} & \\
(\lambda(x)a)(v) & \longrightarrow & a\{v/x\} & \\
\texttt{let}\ x = v\ \texttt{in}\ a & \longrightarrow & a\{v/x\} &
\end{array}
$$

On remarquera que la règle de redéfinition de méthode permettrait également l'extension. C'est le typage qui l'interdira. Un autre point est que la liste des méthodes publiques $\mathcal{P}$ attachée à chaque objet est ignorée par les règles de sémantique, et ne sert que pour le typage.

(Objet-Vide)
$$\frac{}{\Gamma \vdash \emptyset : (\omega, \tau)}$$

(Objet-Méthode)
$$\frac{\Gamma \vdash L : (\omega, \tau) \qquad \Gamma; x : \mathit{Obj}(\omega, \tau) \vdash a : \tau' \qquad \omega = (l : \tau'; \omega')}{\Gamma \vdash L; (l = \varsigma(x)a) : (\omega, \tau)}$$

(Objet)
$$\frac{\Gamma \vdash L : (\mathcal{L}, \tau) \qquad \operatorname{dom} \mathcal{L} = \operatorname{dom} L \qquad \tau = \langle \mathcal{L}|_{\mathcal{P}} \rangle}{\Gamma \vdash [L]_{\mathcal{P}} : \mathit{Obj}(\mathcal{L}, \tau)}$$

(Masquage)
$$\frac{\Gamma \vdash a : \mathit{Obj}(\omega, \tau)}{\Gamma \vdash a : \tau}$$

(Sélection)
$$\frac{\Gamma \vdash a : \langle l : \tau; \omega \rangle}{\Gamma \vdash a\#l : \tau}$$

(Sélection-Privée)
$$\frac{\Gamma \vdash a : \mathit{Obj}((l : \tau'; \omega), \tau)}{\Gamma \vdash a\#l : \tau'}$$

(Redéfinition)
$$\frac{\Gamma \vdash a : \mathit{Obj}(\omega, \tau) \qquad \Gamma; x : \mathit{Obj}(\omega, \tau) \vdash a' : \tau' \qquad \omega = (l : \tau'; \omega')}{\Gamma \vdash a\#l \Leftarrow \varsigma(x)a' : \mathit{Obj}(\omega, \tau)}$$

Fig. 2.1: Règles de typage des objets

Règles de typage

Comme nous en discutions en section 1.5.2 (page 25), nous allons donc donner deux sortes de types aux objets. Un type d'objet *complet* est ajoutée à la grammaire des types :

$$\tau ::= \alpha \mid \tau \to \tau \mid \mu(\alpha)\tau \mid \langle \omega \rangle \mid \mathit{Obj}(\omega, \tau) \qquad \text{Type}$$
$$\omega ::= \rho \mid \emptyset \mid l : \tau; \omega$$

Ce type est composé d'une rangée de types ω et d'un type τ. La rangée ω donne le type de toutes les méthodes de l'objet, tandis que le type τ ne contient que les méthodes publiques. Une règle de typage permet de transformer ce type complet en le type d'objet τ.

Les règles de sous-typage (section 1.4.5 page 23) sont inchangées. Deux types complets ne sont donc en relation de sous-typage que s'ils sont équivalent. Ainsi, ces types peuvent être utilisés pour le typage des redéfinitions de méthodes. En plus du jugement $\Gamma \vdash a : \tau$, le jugement $\Gamma \vdash L : (\omega, \tau)$ est utilisé pour typer le corps des objets. Les règles de typage sont présentées dans la figure 2.1. Le corps d'un objet est typé récursivement en partant du corps vide (règle Objet-Vide). Afin de typer la méthode l d'un objet, on type son corps a en supposant que le type de la variable x est le type de l'objet lui-même (règle Objet-Méthode). La redéfinition d'une méthode (règle Redéfinition) est typée de manière similaire. Pour que l'on puisse donner un type à un objet (règle Objet), le type de son corps doit être non extensible, c'est-à-dire une paire $\mathit{Obj}(\mathcal{L}, \tau)$ où $\mathcal{L}$ est une rangée bornée ($\mathcal{L} ::= \emptyset \mid (l : \tau; \mathcal{L})$). De plus, toutes les méthodes de la rangée $\mathcal{L}$ doivent être définies. Le type τ de l'objet est alors donné par la restriction de $\mathcal{L}$

à l'ensemble $\mathcal{P}$ des méthodes publiques. Contrairement à cette règle, les règles précédentes autorisent des rangées ω non nécessairement bornées, car elles sont également utilisées pour le typage du corps des prototypes, qui sont extensibles. La règle MASQUAGE permet de cacher la partie privée ω du type complet d'un objet. Un appel de méthode peut être typé de deux manières différentes suivant que l'on connaisse le type complet de l'objet (règle SÉLECTION-PRIVÉE) ou non (règle SÉLECTION). Le type du résultat est obtenu a partir de l'un ou l'autre de ces types suivant le cas.

Exemple

L'exemple suivant permet d'illustrer les règles de typage. Cet objet *point* a une méthode x qui joue le rôle d'une variable d'instance, une méthode *position* qui retourne la valeur de x et une méthode *déplace* qui permet de changer la valeur de x. Seules les deux dernières méthodes sont publiques.

$$\begin{aligned} point ::= [&x = \varsigma(s)7; position = \varsigma(s)s\#x; \\ &déplace = \varsigma(s)\lambda(y)s\#x \Leftarrow \varsigma(s')y]_{\{position, déplace\}} \end{aligned}$$

Le type complet de cet objet est : $Obj((x : int; position : int; déplace : int \to \tau), \tau)$, où $\tau = \mu(\alpha)\langle position : int; déplace : int \to \alpha\rangle$. Le type de l'objet après application de la règle MASQUAGE est τ. Ce type indique que l'objet a une méthode *position* qui retourne un entier et une méthode déplace qui prend en argument un entier et retourne un objet de même type τ.

2.1.2 Prototypes

Syntaxe

Nous présentons maintenant les constructions correspondantes aux prototypes. Un prototype $[L]$ est similaire à un objet, mais certaines de ces méthodes peuvent ne pas être définies. On le transforme en un objet par la construction $\texttt{new}_{\mathcal{P}}\ a$, dans laquelle $\mathcal{P}$ est l'ensemble des noms des méthodes devant être publiques. La construction $a\#l \Leftarrow \varsigma(x)a$ permet de redéfinir une méthode d'un prototype ou d'étendre un prototype. La même syntaxe que pour la redéfinition d'une méthode dans un objet est utilisée, car il n'y a pas d'ambiguïté.

$a ::= \dots$	
$\mid [L]$	Corps de prototype
$\mid a\#l \Leftarrow \varsigma(x)a$	Définition de méthode
$\mid \texttt{new}_{\mathcal{P}}\ a$	Constructeur

Sémantique

Les prototypes sont également des valeurs.

$$v ::= \dots \mid [L]$$

Voici dans quels contextes une expression concernant un prototype peut être réduite :

$$E ::= \ldots \mid \mathtt{new}_{\mathcal{P}}\ E \mid E\#l \Leftarrow \varsigma(x)a \mid E \,/\, l \mid E + a \mid v + E \mid E@(l = l)$$

L'expression $\mathtt{new}_{\mathcal{P}}\ [L]$ s'évalue en un objet dont le corps est le même que celui du prototype et dont les méthodes publiques sont données par $\mathcal{P}$. Étendre (ou modifier) un prototype consiste à rajouter la méthode dans son corps.

$$\begin{array}{rcl} \mathtt{new}_{\mathcal{P}}\ [L] & \longrightarrow & [L]_{\mathcal{P}} \\ [L]\#l \Leftarrow \varsigma(x)a & \longrightarrow & [L; (l = \varsigma(x)a)] \end{array}$$

Règles de typage

La grammaire des types est modifiée afin d'incorporer les types des prototypes. Ceux-ci sont des triplets $Prot(\omega, \tau, \mathcal{D})$, où ω et τ sont les interfaces privées et publiques du prototype (ils correspondent aux types ω et τ d'un type d'objet $Obj(\omega, \tau)$), et $\mathcal{D}$ est l'ensemble des méthodes qu'il contient effectivement :

$$\tau ::= \ldots \mid Prot(\omega, \tau, \mathcal{D})$$

Comme pour les objets, la relation de sous-typage sur les types de prototypes coïncide avec l'identité :

$$\tau \preceq Prot(\omega_1, \tau_1, \mathcal{D}_1) \Longrightarrow \tau = Prot(\omega_1, \tau_1, \mathcal{D}_1)$$

Les règles de typage sont les mêmes pour le corps d'un prototype que pour le corps d'un objet. Nous ne les reprenons donc pas ici. Si le corps d'un prototype a le type (ω, τ), ce prototype a le type $Prot(\omega, \tau, \mathcal{D})$, où $\mathcal{D}$ est la liste des méthodes qu'il définit (règle PROTOTYPE ci-dessous). Si un prototype a le type $Prot(\omega, \tau, \mathcal{D})$, on peut lui ajouter une méthode l à condition que cette méthode apparaisse dans ω avec un type τ' et que le corps de cette méthode l ait le type τ' dans l'environnement étendu avec la variable x de type $Obj(\omega, \tau)$ (règle DÉFINITION). Comme la méthode l est alors définie, on obtient un prototype de type $Prot(\omega, \tau, \mathcal{D} \cup \{l\})$. Enfin, on peut créer un objet à partir d'un prototype (règle CRÉATION) à condition que toutes ses méthodes soient définies. Le type τ de l'objet est la partie publique du type du prototype.

(PROTOTYPE)
$$\frac{\Gamma \vdash L : (\omega, \tau) \qquad \mathcal{D} = \operatorname{dom} L}{\Gamma \vdash [L] : Prot(\omega, \tau, \mathcal{D})}$$

(DÉFINITION)
$$\frac{\Gamma \vdash a : Prot(\omega, \tau, \mathcal{D}) \qquad \Gamma; x : Obj(\omega, \tau) \vdash a' : \tau' \qquad \omega = (l : \tau'; \omega')}{\Gamma \vdash (a\#l \Leftarrow \varsigma(x)a') : Prot(\omega, \tau, \mathcal{D} \cup \{l\})}$$

(CRÉATION)
$$\frac{\Gamma \vdash a : Prot(\mathcal{L}, \tau, \mathcal{D}) \qquad \mathcal{D} = \operatorname{dom} \mathcal{L} \qquad \tau = \langle \mathcal{L}|_{\mathcal{P}} \rangle}{\Gamma \vdash \mathtt{new}_{\mathcal{P}}\ a : \tau}$$

Exemples

Considérons maintenant quelques exemples. La classe suivante peut être encodée dans le calcul :

```
class point x0 =
  objet
    val mutable x = x0
    method position = x
    method déplace y = x <- y
  end;;
```

Elle s'écrit sous la forme d'une fonction retournant un prototype. La variable d'instance est représentée comme une méthode constante :

$$cl\_point ::= \lambda(x_0)[x = \varsigma(s)x_0; position = \varsigma(s)s\#x; \\ déplace = \varsigma(s)\lambda(y)s\#x \Leftarrow \varsigma(s')y]$$

Le type le plus général pour cette fonction est :

$$\forall(\alpha)\forall(\beta)\forall(\rho)(\alpha \rightarrow Prot((x : \alpha; position : \alpha; déplace : \alpha \rightarrow \beta; \rho), \\ \beta, \{x, position, déplace\}))$$

Le prototype a des méthodes x et *position*, toutes deux de type α, et une méthode *déplace* qui prend un argument du même type et retourne un objet de type β, qui est le type des objets dérivés du prototype (deuxième composant du triplet). Enfin, ces trois méthodes constituent l'ensemble des méthodes définies (troisième composant).

Le codage que nous venons de donner ne correspond pas exactement à la classe `point`. En effet, lorsque l'on crée un objet de la classe *cl_point*, on peut choisir librement lesquelles des trois méthodes sont publiques car le type β apparaissant dans le type du prototype peut être instancié à n'importe quelle valeur. Mais on peut forcer les méthodes *position* et *déplace* à être publique en spécialisant β en le type $\mu(\beta)\langle position : \alpha; déplace : \alpha \rightarrow \beta; \rho'\rangle$, En effet, la règle de typage Création impose que les méthodes apparaissant dans β soient publiques.

La construction suivante s'évalue en un point dont la position est 7.

$$\mathtt{new}_{\{position, déplace\}}\ cl\_point(7)$$

La méthode x est privée (ce que l'on attend d'une variable d'instance) et les deux autres sont publiques, comme on le voit dans le type de cet objet :

$$\mu(\alpha)\langle position : int; déplace : int \rightarrow \alpha\rangle$$

On peut également créer une classe *cl_point_coloré* étendant la classe précédente avec une méthode *couleur* :

$$cl\_point\_coloré ::= \lambda(x_0)\lambda(c_0)cl\_point(x_0)\#couleur \Leftarrow \varsigma(s)c_0$$

et créer un point coloré :

$$\mathtt{new}_{\{position, déplace, couleur\}}\ cl\_point(11)("rouge")$$

2.1.3 Extensions

Nous présentons maintenant quelques constructions supplémentaires qui ne sont pas essentielles mais permettent d'augmenter l'expressivité du calcul.

Syntaxe

L'expression $a+a'$ dénote la fusion des prototypes a et a'. La construction a / l permet d'oublier la définition de la méthode l, et donc de la rendre abstraite. Enfin, la construction $a@(l = l')$ ajoute la méthode l au prototype a en lui donnant la même définition que la méthode l'. Ainsi, il est possible de conserver l'accès à cette définition même si la méthode l est modifiée par la suite.

$$
\begin{array}{lr}
a ::= \ \dots & \\
\quad \mid a + a & \text{Fusion de prototypes} \\
\quad \mid a \ / \ l & \text{Oubli de la définition d'une méthode} \\
\quad \mid a@(l = l) & \text{Duplication de méthode}
\end{array}
$$

Sémantique

Les contextes d'évaluation doivent être étendus afin de prendre en compte ces nouvelles constructions :

$$E ::= \ \dots \mid E \ / \ l \mid E + a \mid v + E \mid E@(l = l)$$

Voici maintenant les règles de réduction. Le corps du prototype obtenu par fusion de deux prototypes est la concaténation des corps L et L' de ceux-ci (en donnant la priorité aux méthodes de L' en cas de conflit). Oublier la définition d'une méthode consiste à l'oublier du corps du prototype. Enfin, l'expression $[L]@(l = l')$ s'évalue en ajoutant la méthode l au corps du prototype, en reprenant la définition de la méthode l'.

$$
\begin{array}{rcll}
[L]@(l = l') & \longrightarrow & [L; (l = \varsigma(x)a)] & \text{si } L(l') = \varsigma(x)a \\
{[L]} \ / \ l & \longrightarrow & [L|_{\mathrm{dom}\, L \setminus \{l\}}] & \\
{[L]} + [L'] & \longrightarrow & [L; L'] &
\end{array}
$$

Règles de typage

Lorsque la définition d'une méthode est oubliée (règle OUBLI), le nom de cette méthode est enlevé de l'ensemble $\mathcal{D}$ des méthodes définies. Pour pouvoir définir une méthode l en reprenant la définition d'une méthode l' (règle COPIE), ces deux méthodes doivent avoir le même type dans ω. On obtient alors un prototype dans lequel la méthode l est définie (elle est ajoutée à $\mathcal{D}$). Deux prototypes peuvent être fusionnés (règle FUSION) s'ils ont les mêmes types ω et τ. Les méthodes définies dans le prototype résultat sont la réunion des méthodes définies dans chacun de ces prototypes.

$$\frac{\Gamma \vdash a : Prot(\omega, \tau, \mathcal{D}) \qquad l \in \mathcal{D}}{\Gamma \vdash (a \;/\; l) : Prot(\omega, \tau, \mathcal{D} \setminus \{l\})} \quad (\textsc{Oubli})$$

$$\frac{\Gamma \vdash a : Prot(\omega, \tau, \mathcal{D}) \qquad \omega = (l : \tau'; \omega_1) = (l' : \tau'; \omega_2) \qquad l' \in \mathcal{D}}{\Gamma \vdash (a@(l = l')) : Prot(\omega, \tau, \mathcal{D} \cup \{l\})} \quad (\textsc{Copie})$$

$$\frac{\Gamma \vdash a : Prot(\omega, \tau, \mathcal{D}) \qquad \Gamma \vdash a' : Prot(\omega, \tau, \mathcal{D}')}{\Gamma \vdash (a + a') : Prot(\omega, \tau, \mathcal{D} \cup \mathcal{D}')} \quad (\textsc{Fusion})$$

Exemples

Nous allons maintenant illustrer sur un exemple comment la construction $a + a'$ permet de modéliser l'héritage multiple. Considérons le prototype :

$$cl\_comparable ::= [compare = \varsigma(s)\lambda(o)(s\#position = o\#position)]$$

La méthode *compare* teste si la méthode *position* d'un objet issu de ce prototype et la méthode *position* de son argument renvoient le même résultat. Le type le plus général de ce prototype est :

$$\begin{array}{l} \forall(\alpha)\forall(\beta)\forall(\rho)\forall(\rho') \\ \quad Prot((position : \alpha; compare : \langle position : \alpha; \rho' \rangle \to bool; \rho), \\ \qquad \beta, \{compare\}) \end{array}$$

D'après ce type, le prototype a une méthode abstraite *position* de type α et une méthode *compare* prenant en argument un objet ayant également une méthode *position* de type α et retournant un booléen. Comme l'une des méthodes est abstraite, on ne peut pas créer directement d'objet à partir de ce prototype.

Il est possible d'écrire une classe *cl_point_comparable* qui hérite à la fois de la classe *cl_point* présentée en section 2.1.2 (page 32 et de la classe *cl_comparable* :

$$cl\_point\_comparable ::= \lambda(x_0)((cl\_point(x_0)) + cl\_comparable)$$

La méthode *position* est maintenant définie car elle est héritée de la classe *cl_point*. On peut donc créer une instance de cette classe :

$$\mathtt{new}_{\{position, déplace, compare\}}\ cl\_point\_comparable(7)$$

Le type de cette instance est :

$$\forall(\rho)\mu(\beta)\langle position : int; déplace : int \to \beta; compare : \langle position : int; \rho \rangle \to bool \rangle$$

On peut aussi lui donner le type plus spécifique suivant, comprenant une méthode binaire :

$$\mu(\beta)\langle position : int; déplace : int \to \beta; compare : \beta \to bool \rangle$$

Il est donc possible de comparer deux « points comparables ».

Voici maintenant un exemple utilisant les deux autres constructions. On va définir une classe héritant de la classe *cl_point*, définissant la méthode *ancien_ déplace* par la même définition que la méthode *déplace* et rendant cette dernière méthode non définie :

$$\lambda(x_0)((cl\_point(x_0))@(ancien\_\ déplace = déplace) \ / \ déplace)$$

Le type de cette fonction est :

$$\begin{aligned}\forall(\alpha)\forall(\beta)\forall(\rho)(\alpha \to Prot((&x : \alpha; position : \alpha; déplace : \alpha \to \beta;\\ &ancien\_\ déplace : \alpha \to \beta; \rho),\\ &\beta, \{x, position, ancien\_\ déplace\}))\end{aligned}$$

Ainsi, les sous-classes de cette classe devront fournir une nouvelle définition de la méthode *déplace* pour que l'on puisse les instancier mais pourront encore utiliser l'ancienne définition via la méthode *ancien_ déplace*.

2.2 Le calcul d'objets

Dans le calcul présenté dans la section précédente, toutes les méthodes d'une classe sont visibles dans ses sous-classes. Il est cependant important en pratique de pouvoir limiter la portée de certaines méthodes. C'est ce que permet le calcul que nous allons maintenant décrire.

2.2.1 Syntaxe

La syntaxe du calcul est donnée en figure 2.2. Nous ne décrivons que les changements par rapport au calcul précédent. On peut distinguer deux sortes de nom de méthodes. Le nom *interne* d'une méthode est celui sous lequel elle est listée dans le corps L d'un objet. Un nom *externe* est un nom sous lequel elle est connue à l'extérieur de l'objet où dans le corps d'une autre méthode. Un *dictionnaire* φ [30] est une fonction injective assurant la traduction des noms externes en noms internes. Un objet $[L]_{\varphi}^{\varphi'}$ a maintenant deux dictionnaires. Le dictionnaire *statique* φ' est fixe et donne accès aux méthodes publiques de l'objet. Le dictionnaire *dynamique* φ est modifié à chaque appel de méthode : il contient alors les méthodes accessibles dans le corps de la méthode appelée, et qui peuvent être privées. Une définition de méthode $\varsigma(x, \varphi'')a$ contient le dictionnaire φ'' devant remplacer ce dictionnaire φ'. Un appel de méthode peut se faire en utilisant l'un ou l'autre des dictionnaires. L'appel $a\#l$ utilise le dictionnaire statique φ'. Il correspond donc à l'invocation d'une méthode publique. L'appel $a.l$ utilise le dictionnaire dynamique φ et permet l'invocation de méthodes privées. La redéfinition de méthode $(a.l \Leftarrow \varsigma(x)a')$ se fait toujours par le dictionnaire dynamique φ. Un prototype $[L]_{\varphi}$ ne porte qu'un dictionnaire. Ce dictionnaire indique quelles méthodes sont visibles de l'extérieur. Contrairement au calcul précédent, l'ajout d'une méthode l à un prototype est explicite $(a + l)$, car il nécessite une modification de ce dictionnaire. La méthode est initialement abstraite. Enfin, une nouvelle construction $a \backslash l$ permet de masquer une méthode privée l.

2.2.2 Sémantique

Les prototypes et les objets sont considérés modulo le renommage de leurs méthodes internes. Soit φ_0 une application bijective des noms des méthodes vers

$a ::= x$	Variable
$\mid \lambda(x)a$	Abstraction
$\mid a(a)$	Application
$\mid \texttt{let}\ x = a\ \texttt{in}\ a$	Définition
$\mid (a : \tau :> \tau)$	Contrainte de sous-typage
$\mid [L]_\varphi^\varphi$	Objet
$\mid a\#l$	Appel de méthode
$\mid a.l$	Appel de méthode interne
$\mid a.l \Leftarrow \varsigma(x)a$	(Re)définition de méthode
$\mid [L]_\varphi$	Prototype
$\mid \texttt{new}_p\ a$	Constructeur
$\mid a + l$	Ajout de méthode
$\mid a \setminus l$	Masquage de méthode
$\mid a\ /\ l$	Oubli de la définition d'une méthode
$\mid a + a$	Fusion de prototypes
$\mid a@(l = l)$	Duplication de méthode
$L ::= \emptyset$	Objet vide
$\mid L; (l = \varsigma(x, \varphi)a)$	Méthode

Fig. 2.2: Syntaxe du calcul

Appel de méthode ($L(\varphi'(l)) = \varsigma(x, \varphi'')a$)

$$[L]_{\varphi}^{\varphi'}\#l \quad \longrightarrow \quad a\{[L]_{\varphi''}^{\varphi'}/x\}$$

Appel de méthode interne ($L(\varphi(l)) = \varsigma(x, \varphi'')a$)

$$[L]_{\varphi}^{\varphi'}.l \quad \longrightarrow \quad a\{[L]_{\varphi''}^{\varphi'}/x\}$$

Redéfinition de méthode

$$[L]_{\varphi}^{\varphi'}.l \Leftarrow \varsigma(x)a \quad \longrightarrow \quad [L; (\varphi(l) = \varsigma(x, \varphi)a)]_{\varphi}^{\varphi'}$$

FIG. 2.3: Règles de réduction : objets

les noms de méthodes. Le renommage des méthodes internes d'un objet $[L]_{\varphi}^{\varphi'}$ (resp. d'un prototype $[L]_{\varphi}$) est $[\varphi_0(L)]_{\varphi_0 \circ \varphi}^{\varphi_0 \circ \varphi'}$ (resp. $[\varphi_0(L)]_{\varphi_0 \circ \varphi}$), où $\varphi_0(L)$ est défini par :

$$\begin{aligned} \varphi_0(\emptyset) &= \emptyset \\ \varphi_0(L; (l = \varsigma(x, \varphi)a)) &= \varphi_0(L); (\varphi_0(l) = \varsigma(x, \varphi_0 \circ \varphi)a) \end{aligned}$$

Ainsi, les prototypes suivants sont équivalents :

$$[l = \varsigma(x, (l_1 = l))a]_{(l_2 = l)} \qquad [l' = \varsigma(x, (l_1 = l'))a]_{(l_2 = l')}$$

La sémantique du calcul est définie de la même manière que précédemment. Les valeurs sont les fonctions, les objets et les prototypes :

$$v ::= \lambda(x)a \mid [L]_{\varphi}^{\varphi} \mid [L]_{\varphi}$$

Les contextes d'évaluation sont définis par la grammaire suivante :

$$\begin{aligned} E ::= {} & [\_] \\ & \mid E(a) \\ & \mid v(E) \\ & \mid \mathtt{let}\ x = E\ \mathtt{in}\ a \\ & \mid E\#l \\ & \mid E.l \\ & \mid E.l \Leftarrow \varsigma(x)a \\ & \mid \mathtt{new}_p\ E \\ & \mid E \setminus l \\ & \mid E\ /\ l \\ & \mid E@(l = l) \\ & \mid E + a \\ & \mid v + E \end{aligned}$$

Création d'un objet

$$\texttt{new}_{\mathcal{P}}\ [L]_\varphi \quad\longrightarrow\quad [L]_\varphi^{\varphi|_{\mathcal{P}}}$$

Ajout de méthode ($l \notin \operatorname{dom}\varphi$ et $l' \notin \operatorname{im}\varphi \cup \operatorname{dom} L$)

$$[L]_\varphi + l \quad\longrightarrow\quad [L]_{\varphi;(l=l')}$$

Définition de méthode

$$[L]_\varphi.l \Leftarrow \varsigma(x)a \quad\longrightarrow\quad [L;(\varphi(l)=\varsigma(x,\varphi)a)]_\varphi$$

Masquage de méthode

$$[L]_\varphi \setminus l \quad\longrightarrow\quad [L]_{\varphi|_{\operatorname{dom}\varphi\setminus\{l\}}}$$

FIG. 2.4: Règles de réduction : prototypes

Oubli de la définition d'une méthode

$$[L]_\varphi\ /\ l \quad\longrightarrow\quad [L|_{\operatorname{dom} L\setminus\{\varphi(l)\}}]_\varphi$$

Duplication d'une méthode ($L(\varphi(l')) = \varsigma(x,\varphi')a$)

$$[L]_\varphi@(l = l') \quad\longrightarrow\quad [L;(\varphi(l)=\varsigma(x,\varphi')a)]_\varphi$$

Combinaison de deux prototypes (on choisit un représentant tel que
$\forall l \in \operatorname{dom}\varphi \cap \operatorname{dom}\varphi' \cdot \varphi(l) = \varphi'(l)$ et
$(\operatorname{im}\varphi \cup \operatorname{dom} L) \cap (\operatorname{im}\varphi' \cup \operatorname{dom} L') = \varphi(\operatorname{dom}\varphi \cap \operatorname{dom}\varphi')$)

$$[L]_\varphi + [L']_{\varphi'} \quad\longrightarrow\quad [L;L']_{\varphi;\varphi'}$$

FIG. 2.5: Règles de réduction : complément sur les prototypes

Contrainte de sous-typage

$$(a : \tau :> \tau') \quad \longrightarrow \quad a$$

Application

$$(\lambda(x)a)(v) \quad \longrightarrow \quad a\{v/x\}$$

Définition

$$\texttt{let}\ x = v\ \texttt{in}\ a \quad \longrightarrow \quad a\{v/x\}$$

FIG. 2.6: Règles de réduction : calcul de base

La relation de réduction locale est donnée par les figures 2.3, 2.4, 2.5 et 2.6. Lors d'un appel de méthode $[L]_{\varphi}^{\varphi'}\#l$ ou $[L]_{\varphi}^{\varphi'}.l$, la définition de la méthode $\varsigma(x, \varphi'')a$ est tirée du corps de l'objet L après traduction du nom de la méthode par l'un ou l'autre des dictionnaires. La variable x est remplacée par l'objet, dont le dictionnaire dynamique devient φ''. La redéfinition d'une méthode $[L]_{\varphi}^{\varphi'}.l \Leftarrow \varsigma(x)a$ consiste à placer la nouvelle définition dans le corps de l'objet, après traduction du nom de méthode par le dictionnaire φ.

Lors de la création d'un objet à partir d'un prototype avec les méthodes publiques $\mathcal{P}$, le dictionnaire statique de l'objet est la restriction du dictionnaire φ du prototype à ses méthodes publiques. Pour ajouter une méthode l à un prototype, on choisit un nouveau nom de méthode l' et on étend le dictionnaire φ du prototype afin qu'il associe l' à l. La définition (ou redéfinition) d'une méthode d'un prototype se fait en ajoutant la définition de la méthode dans le corps du prototype après traduction de son nom par le dictionnaire du prototype. Une méthode est masquée en la retirant du dictionnaire du prototype.

Afin d'oublier la définition d'une méthode, on l'enlève du corps du prototype. La duplication d'une méthode s'effectue en ajoutant dans le corps du prototype la nouvelle méthode l' en reprenant le corps de l'autre méthode l. Deux prototypes peuvent être fusionnés si leurs dictionnaires φ et φ' coïncident sur leur domaine commun, et si les méthodes internes communes aux deux prototypes sont exactement les méthodes devant être partagées entre les deux prototypes (car elles ont le même nom externe). Il existe toujours un représentant de chacun des prototypes remplissant ces conditions (théorème 10, page 55). Le corps du prototype obtenu est alors la concaténation des corps des prototypes fusionnés, et son dictionnaire est la réunion de leurs dictionnaires.

La relation de réduction locale $\longrightarrow$ est étendue en une relation définissant un pas d'évaluation : $a \longrightarrow a'$ ssi il existe des expressions a_1 et a_1' telles que $a = E[a_1]$, $E[a_1] \longrightarrow E[a_1']$ et $a' = E[a_1']$.

2.2.3 Types

Les types sont quasiment les mêmes que précédemment : la seule différence est que le type complet d'un objet et le type d'un prototype utilisent ici une rangée bornée $\mathcal{L}$ plutôt qu'une rangée ω. Cela reflète le fait que les méthodes sont maintenant introduites explicitement par la construction $a + l$.

$$
\begin{aligned}
\tau &::= \alpha \mid \tau \to \tau \mid \langle\omega\rangle \mid \mu(\alpha)\tau \mid \mathit{Obj}(\mathcal{L}, \tau) \mid \mathit{Prot}(\mathcal{L}, \tau, \mathcal{D}) \\
\omega &::= \rho \mid \emptyset \mid l : \tau; \omega \\
\sigma &::= \forall(\alpha)\sigma \mid \tau
\end{aligned}
$$

Nous rappelons que $\mathcal{D}$ est un ensemble de noms de méthodes, et que les rangées bornées sont définies comme une sous-grammaire des rangées :

$$\mathcal{L} ::= \emptyset \mid l : \tau; \mathcal{L}$$

2.2.4 Règles de typage

Les règles de typage utilisent le jugement $\Gamma \vdash a : \tau$ pour les expressions et le jugement $\Gamma \vdash L : (\mathcal{L}, \tau)$ pour le corps des objets et des prototypes.

Nous reprenons les règles du calcul de base, ainsi que les règles de sous-typage :

$$
\overset{\text{(VAR)}}{\frac{(x : \forall(\vec{\alpha})\tau) \in \Gamma}{\Gamma \vdash x : \tau\{\vec{\tau}/\vec{\alpha}\}}}
\qquad
\overset{\text{(ABS)}}{\frac{\Gamma; x : \tau' \vdash a : \tau}{\Gamma \vdash \lambda(x)a : \tau' \to \tau}}
\qquad
\overset{\text{(APP)}}{\frac{\Gamma \vdash a : \tau' \to \tau \qquad \Gamma \vdash a' : \tau'}{\Gamma \vdash a(a') : \tau}}
$$

$$
\overset{\text{(LET)}}{\frac{\Gamma \vdash a : \tau \qquad \Gamma; x : \operatorname{gen}(\tau, \Gamma) \vdash a' : \tau'}{\Gamma \vdash \mathtt{let}\ x = a\ \mathtt{in}\ a' : \tau'}}
\qquad
\overset{\text{(SOUS-TYPAGE)}}{\frac{\Gamma \vdash a : \tau \qquad \tau \preceq \tau'}{\Gamma \vdash a : \tau'}}
$$

$$
\overset{\text{(CSTR)}}{\frac{\Gamma \vdash a : \theta(\tau) \qquad \tau \preceq \tau'}{\Gamma \vdash (a : \tau :> \tau') : \theta(\tau')}}
$$

Le type d'un objet est formé à partir du type de son corps en composant la rangée bornée $\mathcal{L}$ contenant le type des méthodes et le dictionnaire dynamique de l'objet φ (règle OBJET). On obtient ainsi les méthodes accessibles via ce dictionnaire et leurs types. De plus, toutes les méthodes apparaissant dans $\mathcal{L}$ doivent être définies, et la seconde composante τ du type de l'objet doit être obtenue en composant la rangée $\mathcal{L}$ avec le dictionnaire statique de l'objet : cette composante donne le type des méthodes publiques de l'objet. Le type du corps d'un objet (ou d'un prototype) est une paire $(\mathcal{L}, \tau)$ où $\mathcal{L}$ est une rangée bornée donnant le type des méthodes de l'objet en fonction de leur nom interne et τ est le type externe de l'objet (ce type liste les méthodes publiques de l'objet). Le corps d'un objet est typé récursivement en partant du corps vide (règles OBJET-VIDE et OBJET-MÉTHODE). Le type du corps d'une méthode dans l'environnement Γ dans lequel on a ajouté la variable x doit être le type de cette méthode dans $\mathcal{L}$. La variable x représente l'objet lui-même et son type est $\mathit{Obj}(\mathcal{L} \circ \varphi, \tau)$, où $\mathcal{L} \circ \varphi$ est le type des méthodes vues depuis le corps de la méthode (et donc avant traduction par le dictionnaire dynamique φ).

(Objet)
$$\frac{\Gamma \vdash L : (\mathcal{L}, \tau) \qquad \operatorname{dom} L = \operatorname{dom} \mathcal{L} \qquad \tau = \langle \mathcal{L} \circ \varphi' \rangle}{\Gamma \vdash [L]_{\varphi}^{\varphi'} : \mathit{Obj}(\mathcal{L} \circ \varphi, \tau)}$$

(Objet-Vide)
$$\frac{}{\Gamma \vdash \emptyset : (\mathcal{L}, \tau)}$$

(Objet-Méthode)
$$\frac{\Gamma \vdash L : (\mathcal{L}, \tau) \qquad \Gamma; x : \mathit{Obj}(\mathcal{L} \circ \varphi, \tau) \vdash a : \mathcal{L}(l)}{\Gamma \vdash L; (l = \varsigma(x, \varphi) a) : (\mathcal{L}, \tau)}$$

La règle Masquage permet d'oublier la partie privée $\mathcal{L}$ du type complet d'un objet. L'appel d'une méthode publique est typée comme précédemment (règle Sélection). L'appel à une méthode privée est typée en utilisant la partie privée $\mathcal{L}$ du type complet de l'objet (règle Sélection-Privée). On peut redéfinir une méthode l par une nouvelle définition a' pourvu que le type de cette expression soit le type de la méthode dans $\mathcal{L}$ (règle Redéfinition). Le type de l'objet reste inchangée.

(Masquage)
$$\frac{\Gamma \vdash a : \mathit{Obj}(\mathcal{L}, \tau)}{\Gamma \vdash a : \tau}$$

(Sélection)
$$\frac{\Gamma \vdash a : \langle l : \tau ; \omega \rangle}{\Gamma \vdash a \# l : \tau}$$

(Sélection-Privée)
$$\frac{\Gamma \vdash a : \mathit{Obj}(\mathcal{L}, \tau)}{\Gamma \vdash a.l : \mathcal{L}(l)}$$

(Redéfinition)
$$\frac{\Gamma \vdash a : \mathit{Obj}(\mathcal{L}, \tau) \qquad \Gamma; x : \mathit{Obj}(\mathcal{L}, \tau) \vdash a' : \mathcal{L}(l)}{\Gamma \vdash (a.l \Leftarrow \varsigma(x) a') : \mathit{Obj}(\mathcal{L}, \tau)}$$

Les méthodes d'un prototype ($\operatorname{dom} \mathcal{L}$) sont des méthodes définies dans ce prototype ($\operatorname{dom} L$) ou des méthodes pour lesquelles un emplacement à été réservé dans le dictionnaire du prototype ($\operatorname{im} \varphi$). Le type d'un prototype présente le type de ses méthodes avant traduction par le dictionnaire φ, le type public τ du prototype et la liste des méthodes définies par le prototype et encore visibles via φ (règle Prototype). On peut créer un objet à partir d'un prototype à condition que toutes les méthodes du prototype soient définies et que son type publique τ corresponde aux méthodes publiques de $\mathcal{L}$.

(Prototype)
$$\frac{\Gamma \vdash L : (\mathcal{L}, \tau) \qquad \operatorname{dom} \mathcal{L} = \operatorname{dom} L \cup \operatorname{im} \varphi}{\Gamma \vdash [L]_{\varphi} : \mathit{Prot}(\mathcal{L} \circ \varphi, \tau, \varphi^{-1}(\operatorname{dom} L))}$$

(Constr)
$$\frac{\Gamma \vdash a : \mathit{Prot}(\mathcal{L}, \tau, \mathcal{D}) \qquad \mathcal{D} = \operatorname{dom} \mathcal{L} \qquad \tau = \langle \mathcal{L}|_{\mathcal{P}} \rangle}{\Gamma \vdash \mathtt{new}_{\mathcal{P}}\ a : \tau}$$

Lors de l'ajout d'une méthode abstraite à un prototype (règle Ajout) le type de cette méthode est insérée dans la liste des types des méthodes du prototype. Cette méthode ne doit pas déjà exister. La définition d'une méthode dans un prototype (règle Définition) est similaire à la redéfinition d'une méthode dans un objet. La méthode devient définie et est donc ajoutée à $\mathcal{D}$.

(Ajout)
$$\frac{\Gamma \vdash a : \mathit{Prot}(\mathcal{L}, \tau, \mathcal{D}) \qquad l \notin \operatorname{dom} \mathcal{L}}{\Gamma \vdash (a + l) : \mathit{Prot}((l : \tau'; \mathcal{L}), \tau, \mathcal{D})}$$

(Définition)
$$\frac{\Gamma \vdash a : \mathit{Prot}(\mathcal{L}, \tau, \mathcal{D}) \qquad \Gamma; x : \mathit{Obj}(\mathcal{L}, \tau) \vdash a' : \mathcal{L}(l)}{\Gamma \vdash (a.l \Leftarrow \varsigma(x) a') : \mathit{Prot}(\mathcal{L}, \tau, \mathcal{D} \cup \{l\})}$$

Quand une méthode est masquée (règle MASQUE), elle est enlevée de la liste des méthodes visibles du prototype ainsi que de l'ensemble des méthodes définies. L'oubli de la définition d'une méthode (règle OUBLI) enlève cette méthode de l'ensemble des méthodes définies $\mathcal{D}$. Dans les deux cas, la méthode doit être définie au préalable.

$$\text{(MASQUE)}\quad \frac{\Gamma \vdash a : Prot(\mathcal{L}, \tau, \mathcal{D}) \qquad l \in \mathcal{D}}{\Gamma \vdash (a \setminus l) : Prot(\mathcal{L}|_{\mathrm{dom}\,\mathcal{L} \setminus \{l\}}, \tau, \mathcal{D} \setminus \{l\})} \qquad \text{(OUBLI)}\quad \frac{\Gamma \vdash a : Prot(\mathcal{L}, \tau, \mathcal{D}) \qquad l \in \mathcal{D}}{\Gamma \vdash (a \,/\, l) : Prot(\mathcal{L}, \tau, \mathcal{D} \setminus \{l\})}$$

On peut copier la définition d'une méthode l' vers une méthode l (règle COPIE) à condition qu'elles aient le même type. La méthode l devient alors définie. Deux prototypes peuvent être fusionnés (règle FUSION) si leurs méthodes communes ont le même type. Les méthodes du résultat sont la réunion des méthodes des deux prototypes (même chose pour les méthodes définies).

$$\text{(COPIE)}\quad \frac{\Gamma \vdash a : Prot(\mathcal{L}, \tau, \mathcal{D}) \qquad \mathcal{L}(l) = \mathcal{L}(l')}{\Gamma \vdash (a@(l = l')) : Prot(\mathcal{L}, \tau, \mathcal{D} \cup \{l\})} \qquad \text{(FUSION)}\quad \frac{\begin{array}{c}\Gamma \vdash a : Prot(\mathcal{L}, \tau, \mathcal{D}) \\ \Gamma \vdash a' : Prot(\mathcal{L}', \tau, \mathcal{D}') \\ \forall l \in \mathrm{dom}\,\mathcal{L} \cap \mathrm{dom}\,\mathcal{L}' \cdot \mathcal{L}(l) = \mathcal{L}'(l)\end{array}}{\Gamma \vdash (a + a') : Prot((\mathcal{L}; \mathcal{L}'), \tau, \mathcal{D} \cup \mathcal{D}')}$$

2.2.5 Exemple

Considérons une classe `c` possédant une méthode privée `interne` et une méthode publique `externe` utilisant la méthode privée.

```
class c =
  object (self)
    method private interne = 1
    method externe = self#interne + 1
  end
```

Cette classe peut être encodée dans le calcul comme suit :

$$c ::= (([\emptyset]_\emptyset + interne + externe).interne \Leftarrow \varsigma(x)1).externe \Leftarrow \varsigma(x)x.interne + 1$$

Partant du prototype vide, on lui ajoute deux méthodes *interne* et *externe*. Puis on définit les deux méthodes. La seconde accède à la première par un appel privé. Le type le plus général possible pour cette expression est :

$$Prot((interne : int; exerne : int), \alpha, \{interne, externe\})$$

Les deux méthodes ont le type *int*. Elles ne sont pas publiques et elles sont définies. Un type correspondant mieux à la définition de classe ci-dessus est :

$$Prot((interne : int; exerne : int), \langle externe : int; \alpha\rangle, \{interne, externe\})$$

Il précise que la méthode *externe* doit être publique.

Créer un objet de la classe `c` correspond à créer à partir du prototype c un objet dont la seule méthode visible est *externe* :

$$\texttt{new}_{\{externe\}}\ c$$

Supposons maintenant que l'on veuille masquer complètement la méthode `interne` dans une sous-classe `d` :

```
class d : sig method externe : int end = c
```

On utilise pour cela l'opérateur de masquage d'une méthode d'un prototype :

$$d ::= c \setminus interne$$

Le type du prototype d est alors

$$Prot((externe : int), \alpha, \{externe\})$$

La méthode *interne* a complètement disparu de ce type.

2.2.6 Correction

La preuve de correction s'effectue comme nous l'avons esquissée dans la section 1.3.4 (page 16). Nous commençons par montrer quelques propriétés générales du typage.

Lemme 1 (Décomposition et remplacement) *Si $\Gamma \vdash E[a] : \tau$, alors il existe un type τ' tel que $\Gamma \vdash a : \tau'$ et tel que, de plus, si $\Gamma \vdash a' : \tau'$ alors $\Gamma \vdash E[a'] : \tau$.*

Démonstration. Par induction sur une dérivation de $\Gamma \vdash E[a] : \tau$. Tous les cas sont immédiats. ∎

Lemme 2 (Renforcement de l'environnement) *Si $\Gamma; x : \sigma \vdash a : \tau$ et $\sigma = \sigma'\{\vec{\tau}/\vec{\alpha}\}$ alors $\Gamma; x : \forall(\alpha)\sigma' \vdash a : \tau$.*

Démonstration. Par induction sur une preuve de la première hypothèse. Les cas intéressants concernent l'accès à l'environnement et la généralisation de types. Considérons donc deux de ces cas.

–
$$\frac{(x' : \forall(\vec{\alpha}'')\tau) \in (\Gamma; x : \sigma)}{\Gamma; x : \sigma \vdash x' : \tau\{\vec{\tau}'/\vec{\alpha}''\}}\ (\text{Var})$$

Si $x' \neq x$, le résultat est clair. Sinon, posons $\sigma' = \forall(\vec{\alpha}')\tau'$. On a alors $\forall(\vec{\alpha}'')\tau = \sigma = \sigma'\{\vec{\tau}/\vec{\alpha}\} = \forall(\vec{\alpha}')\tau'\{\vec{\tau}/\vec{\alpha}\}$ (en supposant les variables $\vec{\alpha}'$ distinctes des variables $\vec{\alpha}$). D'où $\tau\{\vec{\tau}'/\vec{\alpha}''\} = \tau'\{\vec{\tau}/\vec{\alpha}\}\{\vec{\tau}'/\vec{\alpha}'\}$.

Donc,
$$\frac{(x : \forall(\vec{\alpha})\forall(\vec{\alpha}')\tau') \in (\Gamma; x : \forall(\alpha)\sigma')}{(\Gamma; x : \forall(\alpha)\sigma') \vdash x : \tau'\{\vec{\tau}/\vec{\alpha}\}\{\vec{\tau}'/\vec{\alpha}'\}}\ (\text{Var})$$

–
$$\frac{\Gamma; x : \sigma \vdash a : \tau \qquad \Gamma; x : \sigma; x' : \mathrm{gen}(\tau, \Gamma; x : \sigma) \vdash a' : \tau'}{\Gamma; x : \sigma \vdash \texttt{let}\ x' = a\ \texttt{in}\ a' : \tau'}\ (\text{Let})$$

L'hypothèse d'induction nous donne

$$\Gamma; x : \forall(\alpha)\sigma' \vdash a : \tau$$

et

$$\Gamma; x : \forall(\alpha)\sigma'; x' : \mathrm{gen}(\tau, \Gamma; x : \sigma) \vdash a' : \tau'$$

Comme $\Gamma; x : \forall(\alpha)\sigma'$ a moins de variables libres que $\Gamma; x : \sigma$, le type $\mathrm{gen}(\tau, \Gamma; x : \forall(\alpha)\sigma')$ est plus général que le type $\mathrm{gen}(\tau, \Gamma; x : \sigma)$. Par conséquent, l'hypothèse d'induction nous donne :

$$\Gamma; x : \forall(\alpha)\sigma'; x' : \mathrm{gen}(\tau, \Gamma; x : \forall(\alpha)\sigma') \vdash a' : \tau'$$

Finalement,

$$\frac{\begin{array}{c}\Gamma; x : \forall(\alpha)\sigma' \vdash a : \tau \\ \Gamma; x : \forall(\alpha)\sigma'; x' : \mathrm{gen}(\tau, \Gamma; x : \forall(\alpha)\sigma') \vdash a' : \tau'\end{array}}{\Gamma; x : \forall(\alpha)\sigma' \vdash \texttt{let } x' = a \texttt{ in } a' : \tau'} \ (\textsc{Let})$$

∎

Lemme 3 (Extension de l'environnement) *Si* $\Gamma \vdash a : \tau$ *et* $\Gamma \subseteq \Gamma'$ *alors* $\Gamma' \vdash a : \tau$.

Démonstration. Par induction sur une preuve de la première hypothèse. La preuve de chaque cas est immédiate. ∎

Lemme 4 (Stabilité par substitution) *Si* $\Gamma \vdash a : \tau$ *alors pour toute substitution* θ, $\theta(\Gamma) \vdash a : \theta(\tau)$.

Démonstration. Par induction sur une preuve de la première hypothèse. Tous les cas sont immédiats à part ceux qui accèdent à l'environnement ou utilisent la généralisation. Considérons donc deux de ces cas.

- $$\frac{(x : \forall(\vec{\alpha})\tau) \in \Gamma}{\Gamma \vdash x : \tau\{\vec{\tau}/\vec{\alpha}\}} \ (\textsc{Var})$$

  Quitte à effectuer un renommage, on peut supposer que les variables $\vec{\alpha}$ ne sont pas des variables libres de Γ et que θ les laisse invariantes.
  Alors $\theta(\forall(\vec{\alpha})\tau) = \forall(\vec{\alpha})\theta(\tau)$ et $\theta(\tau\{\vec{\tau}/\vec{\alpha}\}) = \theta(\tau)\{\theta(\vec{\tau})/\vec{\alpha}\}$.
  On a donc :
  $$\frac{(x : \forall(\vec{\alpha})\theta(\tau)) \in \theta(\Gamma)}{\theta(\Gamma) \vdash x : \theta(\tau)\{\theta(\vec{\tau})/\vec{\alpha}\}} \ (\textsc{Var})$$

  et donc
  $$\theta(\Gamma) \vdash x : \theta(\tau\{\vec{\tau}/\vec{\alpha}\})$$

- $$\frac{\Gamma \vdash a : \tau \qquad \Gamma; x : \mathrm{gen}(\tau, \Gamma) \vdash a' : \tau'}{\Gamma \vdash \texttt{let } x = a \texttt{ in } a' : \tau'} \ (\textsc{Let})$$

  Soit θ_1 une substitution telle que $\theta_1(\Gamma) = \theta(\Gamma)$ et telle que les images des variables généralisables de τ par θ_1 soient des variables distinctes non liées dans $\theta_1(\Gamma)$.

L'hypothèse de récurrence nous donne

$$\theta_1(\Gamma) \vdash a : \theta_1(\tau)$$

et

$$\theta(\Gamma; x : \mathrm{gen}(\tau, \Gamma)) \vdash a' : \theta(\tau')$$

On vérifie que $\theta(\mathrm{gen}(\tau, \Gamma)) = \mathrm{gen}(\theta_1(\tau), \theta_1(\Gamma))$.
Il vient alors

$$\frac{\theta_1(\Gamma) \vdash a : \theta_1(\tau) \qquad \theta_1(\Gamma); x : \mathrm{gen}(\theta_1(\tau), \theta_1(\Gamma)) \vdash a' : \theta(\tau')}{\theta_1(\Gamma) \vdash \texttt{let}\ x = a\ \texttt{in}\ a' : \theta(\tau')}\ (\textsc{Let})$$

et donc

$$\theta(\Gamma) \vdash \texttt{let}\ x = a\ \texttt{in}\ a' : \theta(\tau')$$

■

Lemme 5 (Commutation des méthodes) *La commutation des méthodes d'un objet ou d'un prototype ne change pas son type : si*

$$\Gamma \vdash L; (l = \varsigma(x, \varphi)a); (l' = \varsigma(x', \varphi')a') : (\mathcal{L}, \tau)$$

et $l \neq l'$, alors

$$\Gamma \vdash L; (l = \varsigma(x', \varphi')a'); (l' = \varsigma(x, \varphi)a) : (\mathcal{L}, \tau)$$

Démonstration. Immédiat en utilisant la règle Objet-Méthode. ■

On pourra donc supposer dans la suite des preuves que si $L(l) = \varsigma(x, \varphi)a$, alors L s'écrit $L'; (l = \varsigma(x, \varphi)a)$.

Lemme 6 (Renommage des noms internes des méthodes) *Le renommage des noms internes des méthodes d'un objet ou d'un prototype ne change pas son type.*

Démonstration. Soit φ_0 une bijection des noms de méthodes vers les noms de méthodes. Supposons donc que l'on ait la dérivation suivante :

$$\frac{\dfrac{\vdots}{\Gamma \vdash L : (\mathcal{L}, \tau)} \qquad \mathrm{dom}\,\mathcal{L} = \mathrm{dom}\,L \cup \mathrm{im}\,\varphi}{\Gamma \vdash [L]_\varphi : \mathit{Prot}(\mathcal{L} \circ \varphi, \tau, \varphi^{-1}(\mathrm{dom}\,L))}\ (\textsc{Prototype})$$

On veut montrer la dérivation suivante :

$$\frac{\dfrac{\vdots}{\Gamma \vdash \varphi_0(L) : (\mathcal{L} \circ \varphi_0^{-1}, \tau)} \qquad \begin{array}{c}\mathrm{dom}(\mathcal{L} \circ \varphi_0^{-1}) = \\ \mathrm{dom}(\varphi_0(L)) \cup \mathrm{im}(\varphi_0 \circ \varphi)\end{array}}{\Gamma \vdash [\varphi_0(L)]_{\varphi_0 \circ \varphi} : \mathit{Prot}(\mathcal{L} \circ \varphi, \tau, \varphi^{-1}(\mathrm{dom}\,L))}\ (\textsc{Prototype})$$

On commence donc par montre que si $\Gamma \vdash L : (\mathcal{L}, \tau)$, alors $\Gamma \vdash \varphi_0(L) : (\mathcal{L} \circ \varphi_0^{-1}, \tau)$. On vérifie aisément qu'en effet, si :

$$\frac{}{\Gamma \vdash \emptyset : (\mathcal{L}, \tau)}\ (\textsc{Objet-Vide})$$

alors :

$$\frac{}{\Gamma \vdash \varphi_0(\emptyset) : (\mathcal{L} \circ \varphi_0^{-1}, \tau)} \text{ (Objet-Vide)}$$

et que si :

$$\frac{\Gamma \vdash L : (\mathcal{L}, \tau) \qquad \Gamma; x : Obj(\mathcal{L} \circ \varphi, \tau) \vdash a : \mathcal{L}(l)}{\Gamma \vdash L; (l = \varsigma(x, \varphi)a) : (\mathcal{L}, \tau)} \text{ (Objet-Méthode)}$$

alors :

$$\frac{\Gamma \vdash \varphi_0(L) : (\mathcal{L} \circ \varphi_0^{-1}, \tau) \qquad \Gamma; x\, Obj((\mathcal{L} \circ \varphi_0^{-1}) \circ \varphi_0, \tau) \vdash a : (\mathcal{L} \circ \varphi_0^{-1})(l)}{\Gamma \vdash \varphi_0(L; (l = \varsigma(x, \varphi)a)) : (\mathcal{L} \circ \varphi_0^{-1}, \tau)} \text{ (Objet-Méthode)}$$

Reste à montrer que $\operatorname{dom}(\mathcal{L} \circ \varphi_0^{-1}) = \operatorname{dom}(\varphi_0(L)) \cup \operatorname{im}(\varphi_0 \circ \varphi)$. Calculons :

$$\begin{aligned} \operatorname{dom}(\mathcal{L} \circ \varphi_0^{-1}) &= \varphi_0(\operatorname{dom}(\mathcal{L})) \\ &= \varphi_0((\operatorname{dom} L \cup \operatorname{im} \varphi)) \\ &= \varphi_0((\operatorname{dom} L)) \cup \varphi_0((\operatorname{im} \varphi)) \\ &= \operatorname{dom}(\varphi_0(L)) \cup \operatorname{im}(\varphi_0 \circ \varphi) \end{aligned}$$

On a bien le résultat désiré.

On montre de même que le type d'un objet est préservé par renommage de ses noms de méthodes internes. ∎

Nous allons maintenant montrer que la réduction préserve le typage. Nous commençons pour cela par montrer quelques lemmes techniques.

Lemme 7 (Substitution) *Si* $\Gamma \vdash v : \tau_0$ *et* $\Gamma; x : \sigma \vdash a_1 : \tau_1$ *où* $\sigma = \forall(\vec{\alpha}_i)\tau_0$ *et* $\alpha_i \notin \mathrm{VL}(\Gamma)$ *alors* $\Gamma \vdash a_1\{v/x\} : \tau_1$

Démonstration. Par induction sur une preuve de la deuxième hypothèse $\Gamma; x : \sigma \vdash a_1 : \tau_1$. Nous ne présentons que quelques cas, les autres cas étant similaires.

–

$$\frac{(x' : \forall(\vec{\alpha})\tau) \in (\Gamma; x : \sigma)}{\Gamma; x : \sigma \vdash x' : \tau\{\vec{\tau}/\vec{\alpha}\}} \text{ (Var)}$$

Si $x \neq x'$, on a comme désiré :

$$\frac{(x' : \forall(\vec{\alpha})\tau) \in \Gamma}{\Gamma \vdash x'\{a/x\} : \tau\{\vec{\tau}/\vec{\alpha}\}} \text{ (Var)}$$

Supposons donc $x = x'$. Alors on a en fait :

$$\frac{(x' : \forall(\vec{\alpha}_i)\tau_0) \in \Gamma; x : \sigma}{\Gamma; x : \sigma \vdash x' : \tau_0\{\vec{\tau}/\vec{\alpha}_i\}} \text{ (Var)}$$

Comme les α_i ne sont pas libres dans Γ, une substitution ne modifiant que ces variables laisse cet environnement inchangé. Par le lemme 4, la première hypothèse du lemme donne donc : $\Gamma \vdash v : \tau_0\{\vec{\tau}/\vec{\alpha}_i\}$, c'est-à-dire : $\Gamma \vdash x\{v/x\} : \tau_0\{\vec{\tau}/\vec{\alpha}_i\}$,

–

$$\frac{\Gamma; x : \sigma; x' : \tau' \vdash a : \tau}{\Gamma; x : \sigma \vdash \lambda(x')a : \tau' \to \tau} \text{ (ABS)}$$

D'après le lemme 4, on voit que l'on peut supposer que les variables α_i ne sont pas libres dans τ'. En appliquant le lemme 3 à la première hypothèse du lemme, il vient $\Gamma; x' : \tau' \vdash v : \tau_0$. D'où, en appliquant l'hypothèse d'induction à la prémisse de la règle ci-dessus, $\Gamma; x' : \tau' \vdash a\{v/x\} : \tau$. Finalement :

$$\frac{\Gamma; x' : \tau' \vdash a\{v/x\} : \tau}{\Gamma \vdash (\lambda(x')a)\{v/x\} : \tau' \to \tau} \text{ (ABS)}$$

–

$$\frac{\Gamma; x : \sigma \vdash a : \tau' \to \tau \qquad \Gamma; x : \sigma \vdash a' : \tau'}{\Gamma; x : \sigma \vdash a(a') : \tau} \text{ (APP)}$$

Le résultat voulu est immédiatement obtenu en appliquant l'hypothèse d'induction aux prémisses de cette règle.

–

$$\frac{\Gamma; x : \sigma \vdash a : \tau \qquad \Gamma; x : \sigma; x' : \mathrm{gen}(\tau, (\Gamma; x : \sigma)) \vdash a' : \tau'}{\Gamma; x : \sigma \vdash \texttt{let } x' = a \texttt{ in } a' : \tau'} \text{ (LET)}$$

L'hypothèse de récurrence appliquée à la première prémisse donne : $\Gamma \vdash a\{v/x\} : \tau$.
D'autre part, d'après le lemme 3, on a $\Gamma; x' : \mathrm{gen}(\tau, (\Gamma; x : \sigma)) \vdash v : \tau_0$. D'où, en applicant l'hypothèse de récurrence à la seconde prémisse, $\Gamma; x' : \mathrm{gen}(\tau, (\Gamma; x : \sigma)) \vdash a'\{v/x\} : \tau'$ Et donc, comme $\mathrm{gen}(\tau, \Gamma)$ est plus général que $\mathrm{gen}(\tau, (\Gamma; x : \sigma))$, le lemme 2 nous donne $\Gamma; x' : \mathrm{gen}(\tau, \Gamma) \vdash a'\{v/x\} : \tau'$.
Finalement,

$$\frac{\Gamma \vdash a\{v/x\} : \tau \qquad \Gamma; x' : \mathrm{gen}(\tau, \Gamma) \vdash a'\{v/x\} : \tau'}{\Gamma; x : \sigma \vdash (\texttt{let } x' = a \texttt{ in } a')\{v/x\} : \tau'} \text{ (LET)}$$

■

Nous pouvons maintenant montrer que le typage est préservé par réduction.

Théorème 8 (Préservation du typage par réduction) *Si* $\Gamma \vdash a : \tau$ *et* $a \longrightarrow a'$*, alors* $\Gamma \vdash a' : \tau$.

Démonstration. Par induction sur une dérivation de typage et par cas sur la règle de réduction utilisée.

Tout d'abord, d'après le lemme 6, un renommage des noms internes des méthodes d'un objet ou d'un prototype ne change pas son type. On peut donc bien considérer ceux-ci à renommage près.

Grâce à la transitivité et à la réflexivité du sous-typage, on peut supposer sans perte de généralité que, dans la preuve d'un jugement $\Gamma \vdash a : \tau$, l'usage de la règle de sous-typage SOUS-TYPAGE alterne avec l'usage d'une autre règle de typage (ou est absente quand aucun non-trivial sous-typage n'est possible).

Il est également suffisant de ne considérer que les cas où la preuve ne se termine pas par l'utilisation du sous-typage ou de la règle MASQUE. On peut en effet appliquer l'hypothèse d'induction au jugement $\Gamma \vdash a : \tau_0$ obtenu avant application de l'une de ces règles de typage. On obtient alors $\Gamma \vdash a' : \tau_0$ et on peut appliquer ces mêmes règles de typage pour conclure.

Enfin, d'après le lemme 1, il suffit de considérer les règles de réduction locale.

– $[L]^{\varphi'}_{\varphi}\#l \longrightarrow a\{[L]^{\varphi'}_{\varphi''}/x\}$ où $L(\varphi'(l)) = \varsigma(x,\varphi'')a$
On a la dérivation suivante :

$$\dfrac{\dfrac{\tau'' \preceq \langle l:\tau;\omega\rangle \qquad \dfrac{\dfrac{\begin{array}{c}\mathrm{dom}\,L=\mathrm{dom}\,\mathcal{L}\\ \tau''=\langle\mathcal{L}\circ\varphi'\rangle\end{array} \qquad \dfrac{\vdots}{\Gamma\vdash L:(\mathcal{L},\tau'')}}{\Gamma\vdash [L]^{\varphi'}_{\varphi} : Obj(\mathcal{L}\circ\varphi,\tau'')}\;(\textsc{Objet})}{\Gamma\vdash [L]^{\varphi'}_{\varphi}:\tau''}\;(\textsc{Masque})}{\Gamma\vdash[L]^{\varphi'}_{\varphi}:\langle l:\tau;\omega\rangle}\;(\textsc{Sous-Typage})}{\Gamma\vdash [L]^{\varphi'}_{\varphi}\#l:\tau}\;(\textsc{Sélection})$$

précédée de :

$$\dfrac{\dfrac{\vdots}{\Gamma\vdash L':(\mathcal{L},\tau'')} \qquad \dfrac{\vdots}{(2.1)\;\; \Gamma;x:(\mathcal{L}\circ\varphi'',\tau'')\vdash a:\mathcal{L}(\varphi'(l))}}{\Gamma\vdash L:(\mathcal{L},\tau'')}\;(\textsc{Méthode})$$

avec $L = L';(\varphi'(l) = \varsigma(x,\varphi'')a)$.
On en déduit immédiatement :

$$\dfrac{\mathrm{dom}\,L=\mathrm{dom}\,\mathcal{L} \qquad \tau''=\langle\mathcal{L}\circ\varphi'\rangle \qquad \dfrac{\vdots}{\Gamma\vdash L:(\mathcal{L},\tau'')}}{(2.2)\;\; \Gamma\vdash [L]^{\varphi'}_{\varphi''}:Obj(\mathcal{L}\circ\varphi'',\tau'')}\;(\textsc{Objet})$$

En appliquant le lemme 7 aux jugements (2.1) et (2.2), il vient :

$$\Gamma\vdash a\{[L]^{\varphi'}_{\varphi''}/x\}:\mathcal{L}(\varphi'(l))$$

On a d'autre part :

$$\langle\mathcal{L}\circ\varphi'\rangle = \tau'' \preceq \langle l:\tau;\tau'\rangle$$

et donc $\mathcal{L}(\varphi'(l)) \preceq \tau$. On en déduit finalement

$$\dfrac{\dfrac{\vdots}{\Gamma\vdash a\{[L]^{\varphi'}_{\varphi''}/x\}:\mathcal{L}(\varphi'(l))}}{\Gamma\vdash a\{[L]^{\varphi'}_{\varphi''}/x\}:\tau}\;(\textsc{Sous-Typage})$$

– $[L]^{\varphi'}_{\varphi}.l \longrightarrow a\{[L]^{\varphi'}_{\varphi''}/x\}$ où $L(\varphi(l)) = \varsigma(x,\varphi'')a$
On a la dérivation suivante :

$$\dfrac{\dfrac{\begin{array}{c}\mathrm{dom}\,L=\mathrm{dom}\,\mathcal{L}\\ \tau''=\langle\mathcal{L}\circ\varphi'\rangle\end{array} \qquad \dfrac{\vdots}{\Gamma\vdash L:(\mathcal{L},\tau'')}}{\Gamma\vdash [L]^{\varphi'}_{\varphi}:Obj(\mathcal{L}\circ\varphi,\tau'')}\;(\textsc{Objet})}{\Gamma\vdash [L]^{\varphi'}_{\varphi}.l:(\mathcal{L}\circ\varphi)(l)}\;(\textsc{Sélection-Privée})$$

précédée de :

$$\dfrac{\dfrac{\vdots}{\Gamma\vdash L':(\mathcal{L},\tau'')} \qquad \dfrac{\vdots}{(2.3)\;\; \Gamma;x:(\mathcal{L}\circ\varphi'',\tau'')\vdash a:\mathcal{L}(\varphi(l))}}{\Gamma\vdash L:(\mathcal{L},\tau'')}\;(\textsc{Méthode})$$

avec $L = L'; (\varphi(l) = \varsigma(x, \varphi'')a)$.
On en déduit immédiatement :

$$\dfrac{\operatorname{dom} L = \operatorname{dom} \mathcal{L} \qquad \tau'' = \langle \mathcal{L} \circ \varphi' \rangle \qquad \dfrac{\vdots}{\Gamma \vdash L : (\mathcal{L}, \tau'')}}{(2.4)\ \ \Gamma \vdash [L]^{\varphi'}_{\varphi''} : Obj(\mathcal{L} \circ \varphi'', \tau'')}\ (\textsc{Objet})$$

En appliquant le lemme 7 aux jugements (2.3) et (2.4), il vient :

$$\Gamma \vdash a\{[L]^{\varphi'}_{\varphi''}/x\} : (\mathcal{L} \circ \varphi)(l)$$

comme désiré.

– $[L]^{\varphi'}_{\varphi}.l \Leftarrow \varsigma(x)a \longrightarrow [L; (\varphi(l) = \varsigma(x, \varphi)a)]^{\varphi'}_{\varphi}$
On a la dérivation suivante :

$$\dfrac{\dfrac{\vdots}{\Gamma; x : (\mathcal{L} \circ \varphi, \tau) \vdash a : (\mathcal{L} \circ \varphi)(l)} \qquad \dfrac{\vdots}{\Gamma \vdash [L]^{\varphi'}_{\varphi} : Obj(\mathcal{L} \circ \varphi, \tau)}}{\Gamma \vdash ([L]^{\varphi'}_{\varphi}.l \Leftarrow \varsigma(x)a) : Obj(\mathcal{L} \circ \varphi, \tau)}\ (\textsc{Définition})$$

précédée de

$$\dfrac{(2.5)\ \ \operatorname{dom} L = \operatorname{dom} \mathcal{L} \qquad \tau = \langle \mathcal{L} \circ \varphi' \rangle \qquad \dfrac{\vdots}{\Gamma \vdash L : (\mathcal{L}, \tau)}}{\Gamma \vdash [L]^{\varphi'}_{\varphi} : Obj(\mathcal{L} \circ \varphi, \tau)}\ (\textsc{Objet})$$

On en déduit :

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash L : (\mathcal{L}, \tau)} \qquad \dfrac{\vdots}{\Gamma; x : (\mathcal{L} \circ \varphi, \tau) \vdash a : \mathcal{L}(\varphi(l))}}{\Gamma \vdash L; (\varphi(l) = \varsigma(x, \varphi)a) : (\mathcal{L}, \tau)}\ (\textsc{Méthode})$$

Comme $\varphi(l) \in \operatorname{dom} L$, l'égalité (2.5) nous permet de déduire que

$$\operatorname{dom}(L; (\varphi(l) = \varsigma(x, \varphi)a)) = \operatorname{dom} L = \operatorname{dom} \mathcal{L}$$

D'où, finalement :

$$\dfrac{\begin{array}{c}\operatorname{dom}(L; (\varphi(l) = \varsigma(x, \varphi)a)) = \\ \operatorname{dom} \mathcal{L} \\ \tau = \langle \mathcal{L} \circ \varphi' \rangle\end{array} \qquad \dfrac{\vdots}{\Gamma \vdash L; (\varphi(l) = \varsigma(x, \varphi)a) : (\mathcal{L}, \tau)}}{\Gamma \vdash [L; (\varphi(l) = \varsigma(x, \varphi)a)]^{\varphi'}_{\varphi} : Obj(\mathcal{L} \circ \varphi, \tau)}\ (\textsc{Objet})$$

– $\mathtt{new}_{\mathcal{P}}\ [L]_{\varphi} \longrightarrow [L]^{\varphi|_{\mathcal{P}}}_{\varphi}$
On a la dérivation suivante :

$$\dfrac{\begin{array}{c}(2.6)\ \ \varphi^{-1}(\operatorname{dom} L) = \operatorname{dom}(\mathcal{L} \circ \varphi) \\ (2.7)\ \ \tau = \langle (\mathcal{L} \circ \varphi)|_{\mathcal{P}} \rangle\end{array} \qquad \dfrac{\vdots}{\Gamma \vdash [L]_{\varphi} : Prot(\mathcal{L} \circ \varphi, \tau, \varphi^{-1}(\operatorname{dom} L))}}{\Gamma \vdash \mathtt{new}_{\mathcal{P}}\ [L]_{\varphi} : \tau}$$

précédée de :

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash L : (\mathcal{L}, \tau)} \qquad (2.8) \quad \operatorname{dom} \mathcal{L} = \operatorname{dom} L \cup \operatorname{im} \varphi}{\Gamma \vdash [L]_\varphi : \mathit{Prot}(\mathcal{L} \circ \varphi, \tau, \varphi^{-1}(\operatorname{dom} L))} \ (\textsc{Prototype})$$

La dérivation suivante nous permettrait de conclure :

$$\dfrac{\dfrac{\begin{array}{c}\operatorname{dom} L = \operatorname{dom} \mathcal{L} \\ \tau = \langle \mathcal{L} \circ \varphi|_\mathcal{P} \rangle\end{array} \qquad \dfrac{\vdots}{\Gamma \vdash L : (\mathcal{L}, \tau)}}{\Gamma \vdash [L]_\varphi^{\varphi|_\mathcal{P}} : \mathit{Obj}(\mathcal{L} \circ \varphi, \tau)} \ (\textsc{Objet})}{\Gamma \vdash [L]_\varphi^{\varphi|_\mathcal{P}} : \tau} \ (\textsc{Masque})$$

Il suffit de prouver pour cela que $\tau = \langle \mathcal{L} \circ \varphi|_\mathcal{P} \rangle$ et $\operatorname{dom} L = \operatorname{dom} \mathcal{L}$. La première égalité se déduit trivialement de (2.7). Montrons l'autre égalité. D'après (2.8), il suffit de montrer que $\operatorname{im} \varphi \subseteq \operatorname{dom} L$. Soit $x \in \operatorname{im} \varphi$ et y tel que $x = \varphi(y)$. Toujours d'après (2.8), $x \in \operatorname{dom} \mathcal{L}$, et donc $y \in \operatorname{dom}(\mathcal{L} \circ \varphi)$. Alors, d'après l'égalité (2.6), $y \in \varphi^{-1}(\operatorname{dom} L)$. D'où $x = \varphi(y) \in \operatorname{dom} L$ comme désiré.

– $[L]_\varphi + l \longrightarrow [L]_{\varphi;(l=l')}$ où $l \notin \operatorname{dom} \varphi$ et $l' \notin \operatorname{im} \varphi \cup \operatorname{dom} L$.
On a la dérivation suivante :

$$\dfrac{l \notin \operatorname{dom}(\mathcal{L} \circ \varphi) \qquad \dfrac{\dfrac{\vdots}{\Gamma \vdash L : (\mathcal{L}, \tau)} \qquad \operatorname{dom} \mathcal{L} = \operatorname{dom} L \cup \operatorname{im} \varphi}{\Gamma \vdash [L]_\varphi : \mathit{Prot}(\mathcal{L} \circ \varphi, \tau, \varphi^{-1}(\operatorname{dom} L))} \ (\textsc{Prototype})}{\Gamma \vdash ([L]_\varphi + l) : \mathit{Prot}((l : \tau'; (\mathcal{L} \circ \varphi)), \tau, \varphi^{-1}(\operatorname{dom} L))} \ (\textsc{Ajout})$$

La dérivation suivante nous permettrait de conclure :

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash L : (\mathcal{L}, \tau)} \qquad \operatorname{dom}(l' : \tau'; \mathcal{L}) = \operatorname{dom} L \cup \operatorname{im}(\varphi; (l = l'))}{\begin{array}{c}\Gamma \vdash [L]_{\varphi;(l=l')} : \\ \mathit{Prot}((l' : \tau'; \mathcal{L}) \circ (\varphi; (l = l')), \tau, (\varphi; (l = l'))^{-1}(\operatorname{dom} L))\end{array}} \ (\textsc{Prototype})$$

Il suffit vérifier les trois équations suivantes :

$$\begin{array}{rcl} \operatorname{dom}(l' : \tau'; \mathcal{L}) & = & \operatorname{dom} L \cup \operatorname{im}(\varphi; (l = l')) \\ (l' : \tau'; \mathcal{L}) \circ (\varphi; (l = l')) & = & l : \tau'; (\mathcal{L} \circ \varphi) \\ (\varphi; (l = l'))^{-1}(\operatorname{dom} L) & = & \varphi^{-1}(\operatorname{dom} L)) \end{array}$$

Calculons :

$$\begin{array}{rcl} \operatorname{dom}(l' : \tau'; \mathcal{L}) & = & \operatorname{dom} \mathcal{L} \cup \{l'\} \\ & = & \operatorname{dom} L \cup \operatorname{im} \varphi \cup \{l'\} \\ & = & \operatorname{dom} L \cup \operatorname{im}(\varphi; (l = l')) \end{array}$$

$$\begin{array}{rcl} (l' : \tau'; \mathcal{L}) \circ (\varphi; (l = l')) & = & l : \tau'; ((l' : \tau'; \mathcal{L}) \circ \varphi) \\ & = & l : \tau'; (\mathcal{L} \circ \varphi) \text{ car } l' \notin \operatorname{im} \varphi \end{array}$$

Finalement, comme $l' \notin \operatorname{dom} L$ et $l \notin \operatorname{dom} \varphi$, la dernière égalité est vérifiée.

- $[L]_\varphi.l \Leftarrow \varsigma(x)a \longrightarrow [L; (\varphi(l) = \varsigma(x,\varphi)a)]_\varphi$
  On a la dérivation suivante :

$$\dfrac{\dfrac{\vdots}{\Gamma; x : (\mathcal{L}\circ\varphi,\tau) \vdash a : (\mathcal{L}\circ\varphi)(l)} \qquad \dfrac{\begin{array}{c}\dfrac{\vdots}{\Gamma \vdash L : (\mathcal{L},\tau)} \\ \mathrm{dom}\,\mathcal{L} = \mathrm{dom}\,L \cup \mathrm{im}\,\varphi\end{array}}{\Gamma \vdash [L]_\varphi : \mathit{Prot}(\mathcal{L}\circ\varphi, \tau, \varphi^{-1}(\mathrm{dom}\,L))}}{\Gamma \vdash ([L]_\varphi.l \Leftarrow \varsigma(x)a) : \mathit{Prot}(\mathcal{L}\circ\varphi, \tau, \varphi^{-1}(\mathrm{dom}\,L) \cup \{l\})}$$

  On en déduit :

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash L : (\mathcal{L},\tau)} \qquad \dfrac{\vdots}{\Gamma; x : (\mathcal{L}\circ\varphi,\tau) \vdash a : \mathcal{L}(\varphi(l))}}{\Gamma \vdash L; (\varphi(l) = \varsigma(x,\varphi)a) : (\mathcal{L},\tau)}\ (\text{Méthode})$$

  et donc finalement (comme $\varphi(l) \in \mathrm{dom}\,\mathcal{L}$) :

$$\dfrac{\begin{array}{c}\dfrac{\vdots}{\Gamma \vdash L; (\varphi(l) = \varsigma(x,\varphi)a) : (\mathcal{L},\tau)} \\ \mathrm{dom}\,\mathcal{L} = \mathrm{dom}\,L \cup \{\varphi(l)\} \cup \mathrm{im}\,\varphi\end{array}}{\Gamma \vdash [L; (\varphi(l) = \varsigma(x,\varphi)a)]_\varphi : \mathit{Prot}(\mathcal{L}\circ\varphi, \tau, \varphi^{-1}(\mathrm{dom}\,L) \cup \{l\})}$$

- $[L]_\varphi \setminus l \longrightarrow [L]_{\varphi|_{\mathrm{dom}\,\varphi\setminus\{l\}}}$
  On a la dérivation suivante :

$$\dfrac{l \in \varphi^{-1}(\mathrm{dom}\,L) \qquad \dfrac{\begin{array}{c}\dfrac{\vdots}{\Gamma \vdash L : (\mathcal{L},\tau)} \\ \mathrm{dom}\,\mathcal{L} = \mathrm{dom}\,L \cup \mathrm{im}\,\varphi\end{array}}{\Gamma \vdash [L]_\varphi : \mathit{Prot}(\mathcal{L}\circ\varphi, \tau, \varphi^{-1}(\mathrm{dom}\,L))}\ (\text{Prototype})}{\Gamma \vdash ([L]_\varphi \setminus l) : \mathit{Prot}(\mathcal{L}\circ\varphi|_{\mathrm{dom}\,\mathcal{L}\circ\varphi\setminus\{l\}}, \tau, \varphi^{-1}(\mathrm{dom}\,L) \setminus \{l\})}\ (\text{Masque})$$

  Comme $l \in \varphi^{-1}(\mathrm{dom}\,L)$, on a $\varphi(l) \in \mathrm{dom}\,L$, donc $\mathrm{dom}\,L \cup \mathrm{im}\,\varphi = \mathrm{dom}\,L \cup \mathrm{im}\,\varphi|_{\mathrm{dom}\,\varphi\setminus\{l\}}$ et donc $\mathrm{dom}\,\mathcal{L} = \mathrm{dom}\,L \cup \mathrm{im}\,\varphi|_{\mathrm{dom}\,\varphi\setminus\{l\}}$ D'où, comme désiré :

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash L : (\mathcal{L},\tau)} \qquad \mathrm{dom}\,\mathcal{L} = \mathrm{dom}\,L \cup \mathrm{im}\,\varphi|_{\mathrm{dom}\,\varphi\setminus\{l\}}}{\Gamma \vdash [L]_{\varphi|_{\mathrm{dom}\,\varphi\setminus\{l\}}} : \mathit{Prot}(\mathcal{L}\circ\varphi|_{\mathrm{dom}\,\varphi\setminus\{l\}}, \tau, \varphi^{-1}(\mathrm{dom}\,L) \setminus \{l\})}\ (\text{Prototype})$$

- $[L]_\varphi \ / \ l \longrightarrow [L|_{\mathrm{dom}\,L\setminus\{\varphi(l)\}}]_\varphi$
  On a la dérivation suivante :

$$\dfrac{l \in \varphi^{-1}(\mathrm{dom}\,L) \qquad \dfrac{\begin{array}{c}\dfrac{\vdots}{\Gamma \vdash L : (\mathcal{L},\tau)} \\ \mathrm{dom}\,\mathcal{L} = \mathrm{dom}\,L \cup \mathrm{im}\,\varphi\end{array}}{\Gamma \vdash [L]_\varphi : \mathit{Prot}(\mathcal{L}\circ\varphi, \tau, \varphi^{-1}(\mathrm{dom}\,L))}\ (\text{Prototype})}{\Gamma \vdash ([L]_\varphi \ / \ l) : \mathit{Prot}(\mathcal{L}\circ\varphi, \tau, \varphi^{-1}(\mathrm{dom}\,L) \setminus \{l\})}\ (\text{Oubli})$$

Posons $L' = L|_{\mathrm{dom}\,L\setminus\{\varphi(l)\}}$. Comme $L = L'; (\varphi(l) = \varsigma(x,\varphi')a)$ pour un certain dictionnaire φ', on a :

$$\frac{\dfrac{\vdots}{\Gamma \vdash L' : (\mathcal{L},\tau)} \qquad \dfrac{\vdots}{\Gamma; x : (\mathcal{L}\circ\varphi',\tau) \vdash a : \mathcal{L}(\varphi(l))}}{\Gamma \vdash L : (\mathcal{L},\tau)}\ (\textsc{Méthode})$$

D'où, comme désiré :

$$\frac{\dfrac{\vdots}{\Gamma \vdash L' : (\mathcal{L},\tau)} \qquad \mathrm{dom}\,\mathcal{L} = \mathrm{dom}\,L' \cup \mathrm{im}\,\varphi}{\Gamma \vdash [L']_\varphi : \mathit{Prot}(\mathcal{L}\circ\varphi,\tau,\varphi^{-1}(\mathrm{dom}\,L)\setminus\{l\})}\ (\textsc{Prototype})$$

– $[L]_\varphi @(l = l') \longrightarrow [L; (\varphi(l) = \varsigma(x,\varphi')a)]_\varphi$ où $L(\varphi(l')) = \varsigma(x,\varphi')a$
On a la dérivation suivante :

$$\frac{(\mathcal{L}\circ\varphi)(l) = (\mathcal{L}\circ\varphi)(l') \qquad \dfrac{\begin{array}{c}\dfrac{\vdots}{\Gamma \vdash L : (\mathcal{L},\tau)} \\ \mathrm{dom}\,\mathcal{L} = \mathrm{dom}\,L \cup \mathrm{im}\,\varphi\end{array}}{\Gamma \vdash [L]_\varphi : \mathit{Prot}(\mathcal{L}\circ\varphi,\tau,\varphi^{-1}(\mathrm{dom}\,L))}\ (\textsc{Prototype})}{\Gamma \vdash ([L]_\varphi @(l = l')) : \mathit{Prot}(\mathcal{L}\circ\varphi,\tau,\varphi^{-1}(\mathrm{dom}\,L)\cup\{l\})}\ (\textsc{Copie})$$

précédée de :

$$\frac{\dfrac{\vdots}{\Gamma \vdash L' : (\mathcal{L},\tau)} \qquad \dfrac{\vdots}{\Gamma; x : (\mathcal{L}\circ\varphi',\tau) \vdash a : \mathcal{L}(\varphi(l'))}}{\Gamma \vdash L : (\mathcal{L},\tau)}\ (\textsc{Méthode})$$

avec $L = L'; (\varphi(l') = \varsigma(x,\varphi')a)$.
Comme $(\mathcal{L}\circ\varphi)(l) = (\mathcal{L}\circ\varphi)(l')$, il vient :

$$\frac{\dfrac{\vdots}{\Gamma \vdash L : (\mathcal{L},\tau)} \qquad \dfrac{\vdots}{\Gamma; x : (\mathcal{L}\circ\varphi',\tau) \vdash a : \mathcal{L}(\varphi(l))}}{\Gamma \vdash L; (\varphi(l) = \varsigma(x,\varphi')a) : (\mathcal{L},\tau)}\ (\textsc{Méthode})$$

D'où finalement :

$$\frac{\begin{array}{c}\dfrac{\vdots}{\Gamma \vdash L; (\varphi(l) = \varsigma(x,\varphi')a) : (\mathcal{L},\tau)} \\ \mathrm{dom}\,\mathcal{L} = \mathrm{dom}\,L \cup \{\varphi(l)\} \cup \mathrm{im}\,\varphi\end{array}}{\Gamma \vdash [L; (\varphi(l) = \varsigma(x,\varphi')a)]_\varphi : \mathit{Prot}(\mathcal{L}\circ\varphi,\tau,\varphi^{-1}(\mathrm{dom}\,L)\cup\{l\})}\ (\textsc{Prototype})$$

– $[L]_\varphi + [L']_{\varphi'} \longrightarrow [L; L']_{\varphi;\varphi'}$ où (2.9)$\forall l \in \mathrm{dom}\,\varphi \cap \mathrm{dom}\,\varphi' \cdot \varphi(l) = \varphi'(l)$ et (2.10)$(\mathrm{im}\,\varphi \cup \mathrm{dom}\,L) \cap (\mathrm{im}\,\varphi' \cup \mathrm{dom}\,L') = \varphi(\mathrm{dom}\,\varphi \cap \mathrm{dom}\,\varphi')$
On a la dérivation suivante (règle Fusion) :

$$\frac{\begin{array}{c}\dfrac{\vdots}{\begin{array}{c}\Gamma \vdash [L]_\varphi : \\ \mathit{Prot}(\mathcal{L}\circ\varphi,\tau,\varphi^{-1}(\mathrm{dom}\,L))\end{array}} \qquad \dfrac{\vdots}{\begin{array}{c}\Gamma \vdash [L']_{\varphi'} : \\ \mathit{Prot}(\mathcal{L}'\circ\varphi',\tau,\varphi'^{-1}(\mathrm{dom}\,L'))\end{array}} \\ (2.11)\ \ \forall l \in \mathrm{dom}(\mathcal{L}\circ\varphi) \cap \mathrm{dom}(\mathcal{L}'\circ\varphi') \cdot (\mathcal{L}\circ\varphi)(l) = (\mathcal{L}'\circ\varphi')(l)\end{array}}{\begin{array}{c}\Gamma \vdash ([L]_\varphi + [L']_{\varphi'}) : \\ ((\mathcal{L}\circ\varphi; \mathcal{L}'\circ\varphi'),\tau,\varphi^{-1}(\mathrm{dom}\,L)\cup\varphi'^{-1}(\mathrm{dom}\,L'))\end{array}}$$

précédée de :

$$\frac{\dfrac{\vdots}{\Gamma \vdash L : (\mathcal{L}, \tau)} \qquad \operatorname{dom} \mathcal{L} = \operatorname{dom} L \cup \operatorname{im} \varphi}{\Gamma \vdash [L]_\varphi : \mathit{Prot}(\mathcal{L} \circ \varphi, \tau, \varphi^{-1}(\operatorname{dom} L))} \ (\textsc{Prototype})$$

et de :

$$\frac{\dfrac{\vdots}{\Gamma \vdash L' : (\mathcal{L}', \tau)} \qquad \operatorname{dom} \mathcal{L}' = \operatorname{dom} L' \cup \operatorname{im} \varphi'}{\Gamma \vdash [L']_{\varphi'} : \mathit{Prot}(\mathcal{L}' \circ \varphi', \tau, \varphi'^{-1}(\operatorname{dom} L'))} \ (\textsc{Prototype})$$

Nous allons montrer la conclusion de la règle ci-dessous, en en montrant les deux prémisses :

$$\frac{\begin{array}{c}\Gamma \vdash (L; L) : (\mathcal{L}; \mathcal{L}'), \tau) \\ \operatorname{dom}(\mathcal{L}; \mathcal{L}') = \operatorname{dom}(L; L') \cup \operatorname{im}(\varphi; \varphi')\end{array}}{\Gamma \vdash [L; L']_{\varphi;\varphi'} : ((\mathcal{L}; \mathcal{L}') \circ (\varphi; \varphi'), \tau, (\varphi; \varphi')^{-1}(\operatorname{dom}(L; L')))} \ (\textsc{Prototype})$$

Il ne restera alors plus qu'à montrer que le type de $[L; L']_{\varphi;\varphi'}$ est bien égale au type de l'expression initiale.
La preuve de $\Gamma \vdash L; L' : ((\mathcal{L}; \mathcal{L}'), \tau)$ se fait par induction sur la largeur de $\mathcal{L}'$. Si $L' = \emptyset$, alors $L; L' = L$. D'autre part, les règles Objet-Vide et Objet-Méthode restent valables si la rangée $\mathcal{L}'$ est remplacée par la rangée plus large $(\mathcal{L}; \mathcal{L}')$ dans le type de L.
Montrons maintenant que $\operatorname{dom}(\mathcal{L}; \mathcal{L}') = \operatorname{dom}(L; L') \cup \operatorname{im}(\varphi; \varphi')$.

$$\begin{aligned}\operatorname{dom}(\mathcal{L}; \mathcal{L}') &= \operatorname{dom} \mathcal{L} \cup \operatorname{dom} \mathcal{L}' \\ &= (\operatorname{dom} L \cup \operatorname{im} \varphi) \cup (\operatorname{dom} L' \cup \operatorname{im} \varphi') \\ &= \operatorname{dom}(L; L') \cup \operatorname{im} \varphi \cup \operatorname{im} \varphi'\end{aligned}$$

Il ne reste plus qu'à montrer que $\operatorname{im} \varphi \cup \operatorname{im} \varphi' = \operatorname{im}(\varphi; \varphi')$. Clairement, $\operatorname{im}(\varphi; \varphi') \subseteq \operatorname{im} \varphi \cup \operatorname{im} \varphi'$. Montrons l'inclusion réciproque. Soit $l \in \operatorname{im} \varphi \cup \operatorname{im} \varphi'$. Si $l \in \operatorname{im} \varphi'$, alors $l \in \operatorname{im}(\varphi; \varphi')$ comme désiré. Supposons donc maintenant $l \notin \operatorname{im} \varphi'$. Alors, il existe l' tel que $l = \varphi(l')$. Si on avait $l' \in \operatorname{dom} \varphi'$, on aurait $l = \varphi(l') = \varphi'(l')$ et donc $l \in \operatorname{im} \varphi'$ (la règle de réduction ne s'applique en effet que si (2.9) $\forall l \in \operatorname{dom} \varphi \cap \operatorname{dom} \varphi' \cdot \varphi(l) = \varphi'(l)$). Comme ce n'est pas le cas, on a donc $(\varphi; \varphi')(l') = \varphi(l') = l$ et donc bien également $l \in \operatorname{im}(\varphi; \varphi')$ comme désiré.
Montrons maintenant l'égalité des types. On a :

$$\begin{aligned}(\mathcal{L}; \mathcal{L}') \circ (\varphi; \varphi') &= ((\mathcal{L}; \mathcal{L}') \circ \varphi); ((\mathcal{L}; \mathcal{L}') \circ \varphi') \\ &= (\mathcal{L} \circ \varphi); (\mathcal{L}' \circ \varphi')\end{aligned}$$

La première égalité est claire. La seconde vient d'une part du fait que $\operatorname{im} \varphi \subseteq \operatorname{dom} \mathcal{L}$ et $\operatorname{im} \varphi' \subseteq \operatorname{dom} \mathcal{L}'$, et d'autre part des égalités (2.9) et (2.11) qui montrent que $\mathcal{L}$ et $\mathcal{L}'$ coïncident sur l'intersection des images de φ et φ'.
Il ne reste plus qu'à montrer que

$$\varphi^{-1}(\operatorname{dom} L) \cup \varphi'^{-1}(\operatorname{dom} L') = (\varphi; \varphi')^{-1}(\operatorname{dom}(L; L'))$$

soit $\mathrm{dom}(L \circ \varphi) \cup \mathrm{dom}(L' \circ \varphi') = \mathrm{dom}((L; L') \circ (\varphi; \varphi'))$. On a clairement :

$$\begin{aligned}
\mathrm{dom}((L; L') \circ (\varphi; \varphi')) &= \mathrm{dom}((L; L') \circ \varphi) \cup \mathrm{dom}((L; L') \circ \varphi') \\
&= \mathrm{dom}(L \circ \varphi) \cup \mathrm{dom}(L' \circ \varphi) \cup \\
& \mathrm{dom}(L \circ \varphi') \cup \mathrm{dom}(L' \circ \varphi')
\end{aligned}$$

Il suffit donc d'avoir $\mathrm{dom}(L' \circ \varphi) \subseteq \mathrm{dom}(L' \circ \varphi')$ et $\mathrm{dom}(L \circ \varphi') \subseteq \mathrm{dom}(L \circ \varphi)$. Nous n'allons considérer que la première égalité, la seconde se montrant de manière similaire. Soit $l \in \mathrm{dom}(L' \circ \varphi)$. On a

$$\begin{aligned}
\varphi(l) &\in \mathrm{dom}\, L' \cap \mathrm{im}\, \varphi \\
&\in (\mathrm{im}\, \varphi \cup \mathrm{dom}\, L) \cap (\mathrm{im}\, \varphi' \cup \mathrm{dom}\, L') \\
&\in \varphi(\mathrm{dom}\, \varphi \cap \mathrm{dom}\, \varphi') \ \text{(d'après (2.10))}
\end{aligned}$$

Par injection de φ, on a donc $l \in \mathrm{dom}\, \varphi \cap \mathrm{dom}\, \varphi'$. Alors, d'après (2.9), $\varphi(l) = \varphi'(l)$ et donc $l \in \mathrm{dom}(L' \circ \varphi')$ comme désiré.

- $(a : \tau :> \tau') \longrightarrow a$
  On a la dérivation suivante :

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash a : \theta(\tau)} \qquad \tau \preceq \tau'}{\Gamma \vdash (a : \tau :> \tau') : \theta(\tau')} \ (\textsc{Cstr})$$

et donc, comme le sous-typage est préservé par instanciation

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash a : \theta(\tau)} \qquad \theta(\tau) \preceq \theta(\tau')}{\Gamma \vdash a : \theta(\tau')} \ (\textsc{Sous-Typage})$$

- $(\lambda(x)a)(v) \longrightarrow a\{v/x\}$
  On a la dérivation suivante :

$$\dfrac{\dfrac{\vdots}{(2.12)\ \Gamma \vdash v : \tau} \qquad \dfrac{\tau_0' \to \tau_0 \preceq \tau' \to \tau \qquad \dfrac{\dfrac{\vdots}{(2.13)\ \Gamma; x : \tau_0' \vdash a : \tau_0}}{\Gamma \vdash \lambda(x)a : \tau_0' \to \tau_0} \ (\textsc{Abs})}{\Gamma \vdash \lambda(x)a : \tau' \to \tau} \ (\textsc{Sous-Typage})}{\Gamma \vdash (\lambda(x)a)(v) : \tau} \ (\textsc{App})$$

L'inéquation $\tau_0' \to \tau_0 \preceq \tau' \to \tau$ entraîne $\tau' \preceq \tau_0'$ et $\tau_0 \preceq \tau$. La règle Sous-Typage appliquée aux jugements (2.12) et (2.13) donne donc :

$$\begin{gathered}
\Gamma \vdash v : \tau_0' \\
\Gamma; x : \tau_0' \vdash a : \tau
\end{gathered}$$

En appliquant le lemme 7 à ces deux jugements, il vient :

$$\Gamma \vdash a\{v/x\} : \tau$$

- `let` $x = v$ `in` $a \longrightarrow a\{v/x\}$
  On a la dérivation suivante :

$$\frac{\begin{array}{c}\Gamma \vdash v : \tau \\ \Gamma; x : \mathrm{gen}(\tau, \Gamma) \vdash a : \tau'\end{array}}{\Gamma \vdash \texttt{let } x = v \texttt{ in } a : \tau'} \ (\textsc{Let})$$

  En appliquant le lemme 7 aux prémisses de cette règle, il vient :

$$\Gamma \vdash a\{v/x\} : \tau'$$

■

Nous allons maintenant montrer qu'une expression bien typée qui n'est pas une valeur peut être réduite. Nous commençons par un lemme permettant de déterminer la forme d'une valeur en fonction de son type.

Lemme 9 (Type des valeurs) *Le type des valeurs permet de les distinguer :*

- *si* $\vdash v : \tau' \to \tau$ *alors* v *est une fonction* $\lambda(x)a$ *;*
- *si* $\vdash v : \langle\omega\rangle$ *ou* $\vdash v : Obj(\mathcal{L}, \tau)$ *alors* v *est un objet* $[L]_{\varphi}^{\varphi'}$ *;*
- *si* $\vdash v : Prot(\mathcal{L}, \tau, \mathcal{D})$ *alors* v *est un prototype* $[L]_{\varphi}$*.*

Démonstration. Par inspection des règles de typage, on voit que

- Si v est une fonction $\lambda(x)a$, alors son type est de la forme $\tau' \to \tau$;
- Si v est un objet $[L]_{\varphi}^{\varphi'}$ alors son type est de la forme $\langle\omega\rangle$ ou $Obj(\mathcal{L}, \tau)$;
- Si v est un prototype $[L]_{\varphi}$ alors $Prot(\mathcal{L}, \tau, \mathcal{D})$.

Comme on a ainsi considéré tous les cas de valeurs et que les formes des types sont disjoints, on en déduit la forme d'une valeur en fonction de son type. ■

Théorème 10 (Absence de bloquage) *Si* $\vdash a : \tau$ *et* a *n'est pas une valeur, alors il existe une expression* a' *telle que* $a \longrightarrow a'$*.*

Démonstration. Supposons que $\vdash a : \tau$ et a n'est pas une valeur. Considérons un contexte d'évaluation E tel que $a = E[a_0]$ et a_0 n'est pas une valeur. D'après le lemme 1, $\vdash a_0 : \tau'$ pour un certain type τ'. Si le théorème est vrai pour a_0 alors $a_0 \longrightarrow a_0'$ et donc $a \longrightarrow E[a_0']$. On peut donc supposer que le plus grand contexte d'évaluation E tel que $a = E[a_0]$ et a_0 n'est pas une valeur soit le contexte vide.

Par ailleurs, quitte à remplacer τ par un autre type plus précis, on peut supposer que $\vdash a : \tau$ peut être déduit à l'aide d'une dérivation ne se terminant pas par l'application de la règle de sous-typage.

On conclut en considérant les formes possibles de a. Nous ne présentons que quelques cas (les autres sont similaires) :

- $a = x$:
  Cette expression n'est pas typable dans l'environnement vide.
- $a = a_1(a_1')$:
  Nécessairement, a_1 et a_1' sont des valeurs v et v'. Une dérivation de $\vdash a : \tau$ contient la règle suivante :

$$\frac{\vdash v : \tau' \to \tau \qquad \vdash a' : \tau'}{\vdash a(a') : \tau} \ (\textsc{App})$$

D'après le lemme 9, v est donc une fonction. Il existe donc une règle de réduction telle que $a \longrightarrow a'$.

- $a = (a' : \tau :> \tau')$:
  On a $a \longrightarrow a'$.
- $a = a_1\#l$:
  Nécessairement, a_1 est une valeur v. Une dérivation de $\vdash a : \tau$ contient la règle suivante :

$$\frac{\vdash v : \langle l : \tau; \omega\rangle}{\vdash v\#l : \tau}\ (\textsc{Sélection})$$

  D'après le lemme 9, v est donc un objet. D'après son type, il a une méthode l. Il existe donc une règle de réduction telle que $a \longrightarrow a'$.
- $a = a_1 + a_2$:
  Nécessairement, a_1 et a_2 sont des valeurs v_1 et v_2. Une dérivation de $\vdash a : \tau$ contient la règle suivante :

$$\frac{\begin{array}{c}\vdash a : \mathit{Prot}(\mathcal{L}, \tau, \mathcal{D})\\ \vdash a' : \mathit{Prot}(\mathcal{L}', \tau, \mathcal{D}')\\ \forall l \in \operatorname{dom}\mathcal{L} \cap \operatorname{dom}\mathcal{L}' \cdot \mathcal{L}(l) = \mathcal{L}'(l)\end{array}}{\vdash (a + a') : \mathit{Prot}((\mathcal{L}; \mathcal{L}'), \tau, \mathcal{D} \cup \mathcal{D}')}\ (\textsc{Fusion})$$

  D'après le lemme 9, ces valeurs sont donc des prototypes $[L]_\varphi$ et $[L']_{\varphi'}$. Il nous suffit donc de vérifier que les conditions associées à la règle de réduction définissant la fusion de deux prototypes peuvent être satisfaites par un renommage adéquat φ_0 des noms de méthodes internes de l'un des prototype, soit :

$$\forall l \in \operatorname{dom}(\varphi_0 \circ \varphi) \cap \operatorname{dom}\varphi' \cdot (\varphi_0 \circ \varphi)(l) = \varphi'(l)$$

  et

$$(\operatorname{im}(\varphi_0 \circ \varphi) \cup \varphi_0(\operatorname{dom} L)) \cap (\operatorname{im}\varphi' \cup \operatorname{dom} L') = (\varphi_0 \circ \varphi)(\operatorname{dom}(\varphi_0 \circ \varphi) \cap \operatorname{dom}\varphi')$$

  Cette seconde équation s'écrit aussi :

$$\varphi_0(\operatorname{im}\varphi \cup \operatorname{dom} L) \cap (\operatorname{im}\varphi' \cup \operatorname{dom} L') = (\varphi_0 \circ \varphi)(\operatorname{dom}\varphi \cap \operatorname{dom}\varphi')$$

  Pour que la première condition soit remplie, il suffit de choisir φ_0 de façon à ce que $\varphi_0(\varphi(l)) = \varphi'(l)$. Comme φ est injectif, cela est toujours possible. On a alors l'inclusion :

$$\varphi_0(\operatorname{im}\varphi \cup \operatorname{dom} L) \cap (\operatorname{im}\varphi' \cup \operatorname{dom} L') \supseteq (\varphi_0 \circ \varphi)(\operatorname{dom}\varphi \cap \operatorname{dom}\varphi')$$

  Pour avoir l'inclusion contraire, il suffit alors de choisir $\varphi_0(l)$ arbitrairement en dehors de $\operatorname{im}\varphi' \cup \operatorname{dom} L'$ pour tout nom de méthode l tel que $l \in \operatorname{im}\varphi \cup \operatorname{dom} L$ et $l \notin \varphi(\operatorname{dom}\varphi \cap \operatorname{dom}\varphi')$.
- $a = a_1 + l$:
  L'expression a_1 est nécessairement une valeur v. Une dérivation de $\vdash a : \tau$ contient la règle suivante :

$$\frac{\vdash v : \mathit{Prot}(\mathcal{L}, \tau, \mathcal{D}) \qquad l \notin \mathrm{dom}\,\mathcal{L}}{\vdash (v + l) : \mathit{Prot}((l : \tau'; \mathcal{L}), \tau, \mathcal{D})} \ (\textsc{Ajout})$$

D'après le lemme 9, v est donc un prototype $[L]_\varphi$, et on a également la règle suivante :

$$(\textsc{Prototype})\quad \frac{\Gamma \vdash L : (\mathcal{L}', \tau) \qquad \mathrm{dom}\,\mathcal{L}' = \mathrm{dom}\,L \cup \mathrm{im}\,\varphi}{\Gamma \vdash [L]_\varphi : \mathit{Prot}(\mathcal{L}' \circ \varphi, \tau, \varphi^{-1}(\mathrm{dom}\,L))}$$

où $\mathcal{L} = \mathcal{L}' \circ \varphi$.
Comme $\mathrm{dom}\,\mathcal{L}' = \mathrm{dom}\,L \cup \mathrm{im}\,\varphi$, $\mathrm{im}\,\varphi \subseteq \mathrm{dom}\,\mathcal{L}'$, et donc $\mathrm{dom}\,\mathcal{L} = \mathrm{dom}\,\mathcal{L}' \circ \varphi = \mathrm{dom}\,\varphi$. Par conséquent, $l \notin \mathrm{dom}\,\varphi (= \mathrm{dom}\,\mathcal{L})$. La condition associée à la règle de réduction définissant l'ajout de méthode est donc satisfaite et cette règle peut être appliquée.

■

Nous pouvons maintenant conclure :

Théorème 11 (Correction du calcul) *Si $\vdash a : \tau$, alors soit $a\Uparrow$, soit il existe une valeur v telle que $a \longrightarrow^* v$. De plus, $\vdash v : \tau$.*

Démonstration. Si la réduction ne diverge pas, alors il existe une expression a' telle que $a \longrightarrow^* a'$ mais pas d'expression a'' telle que $a' \longrightarrow a''$. D'après le théorème 8, $\vdash a' : \tau$. D'après le théorème 10, a' est une valeur. ■

2.3 Classes

Le calcul présenté dans la section précédente ne serait pas très pratique à utiliser pour programmer. En effet, les constructions qu'il fournit pour créer des objets sont de trop bas niveau. Nous allons donc maintenant l'étendre avec des classes, qui permettent de définir plus aisément des objets. La sémantique de cette extension est définie par traduction dans le calcul non étendu.

2.3.1 Grammaire

Le langage est donc défini comme une extension du calcul présenté dans la section 2.2 page 35. Nous ne présentons ci-dessous que les nouvelles constructions.

$a ::= \ldots$	
$\mid \texttt{new}^{\mathcal{P}}\ a$	Constructeur
$\mid (a : \tau < \tau)$	Contrainte de classe
$\mid \texttt{object}\ (x)\ d\ \texttt{end}$	Classe
$d ::= \emptyset$	
$\mid d; \texttt{abstract}\ l$	Introduction de méthode

$\mid d;$ `method` $l = a$	Définition de méthode
$\mid d;$ `public` l	Exportation de méthode
$\mid d;$ `inherit` a	Héritage
$\mid d;$ `rename` l `as` l	Duplication de méthode

La construction $\texttt{new}^{\mathcal{P}}\ a$ permet de créer un objet à partir d'une classe a. Les méthodes publiques de l'objet doivent être données explicitement par l'ensemble $\mathcal{P}$.

Il est possible de modifier le type d'une classe, d'un type τ à un type τ', par exemple masquer certaines méthodes et en rendre d'autres publiques, à l'aide d'une contrainte de classe $(a : \tau < \tau')$.

Enfin, le corps d d'une classe `object` (x) d `end` décrit les méthodes de ses instances. La variable x permet de faire référence depuis la définition d'une méthode à l'objet auquel elle appartiendra.

Un corps de classe se lit séquentiellement. Une méthode doit d'abord être explicitement introduite à l'aide de la construction `abstract` l. Une définition peut alors être fournie pour cette méthode par la construction `method` $l = a$. La construction `public` l rend la méthode l publique. L'héritage (`inherit` a) consiste à ajouter toutes les méthodes de la classe parente a à la classe courante. Finalement, il est possible de reprendre la définition d'une méthode l' pour définir une méthode l à l'aide de la construction `rename` l' `as` l.

Les types sont étendues avec des types de classes :

$$\tau ::= \ \ldots \mid Cl(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P})$$

Un type de classe est composé des éléments suivants :

- une rangée bornée $\mathcal{L}$ donnant le type des méthodes visibles de la classe ;
- un type τ donnant la forme des types des objets de la classe (leurs types seront une instance de ce type) ;
- l'ensemble $\mathcal{D}$ des noms des méthodes définies ;
- l'ensemble $\mathcal{P}$ des noms des méthodes publiques.

2.3.2 Traduction

La sémantique du langage est définie par une traduction typée vers le calcul défini précédemment. Cette traduction est définie à l'aide de règles d'inférence donnant simultanément le type et la traduction des expressions. Ces règles font usage de trois sortes de jugement : $\Gamma \vdash a : \tau \Rightarrow a_1$ donne le type τ et la traduction a_1 d'une expression a, $\Gamma, x : \tau \vdash d : \tau' \Rightarrow a_1$ donne le type τ' et la traduction a_1 d'un corps de classe d. Enfin, $\vdash \tau < \tau' \Rightarrow a$ définie une fonction de coercion d'un type τ vers un type τ'. Ce dernier jugement est utilisé pour le typage et la traduction des contraintes de classes.

Les figures 2.7 et 2.8 donnent la traduction du calcul de base, qui consiste essentiellement à remplacer dans une expression de ce calcul les sous-expressions par leur traduction. Les figures 2.9 et 2.10 donnent les règles de traduction des classes. Ces règles permettent de traduire les constructions de classes en les constructions correspondantes pour les prototypes. La règle de traduction de la

$$\frac{(x : \forall(\vec{\alpha})\tau) \in \Gamma}{\Gamma \vdash x : \tau\{\vec{\tau}/\vec{\alpha}\} \Rightarrow x} \qquad \frac{\Gamma; x : \tau' \vdash a : \tau \Rightarrow a_1}{\Gamma \vdash \lambda(x)a : \tau' \to \tau \Rightarrow \lambda(x)a_1}$$

$$\frac{\Gamma \vdash a : \tau' \to \tau \Rightarrow a_1 \qquad \Gamma \vdash a' : \tau' \Rightarrow a'_1}{\Gamma \vdash a(a') : \tau \Rightarrow a_1(a'_1)}$$

$$\frac{\Gamma \vdash a : \tau \Rightarrow a_1 \qquad \Gamma; x : \mathrm{gen}(\tau, \Gamma) \vdash a' : \tau' \Rightarrow a'_1}{\Gamma \vdash \texttt{let}\ x = a\ \texttt{in}\ a' : \tau' \Rightarrow \texttt{let}\ x = a_1\ \texttt{in}\ a'_1}$$

$$\frac{\Gamma \vdash a : \tau \Rightarrow a_1 \qquad \tau \preceq \tau'}{\Gamma \vdash a : \tau' \Rightarrow a_1} \qquad \frac{\Gamma \vdash a : \theta(\tau) \Rightarrow a_1 \qquad \tau \preceq \tau'}{\Gamma \vdash (a : \tau :> \tau') : \theta(\tau') \Rightarrow (a_1 : \tau :> \tau')}$$

$$\frac{\begin{array}{c}\Gamma \vdash L : (\mathcal{L}, \tau) \Rightarrow L_1 \\ \mathrm{dom}\, L = \mathrm{dom}\, \mathcal{L} \qquad \tau = \langle \mathcal{L} \circ \varphi' \rangle\end{array}}{\Gamma \vdash [L]_{\varphi}^{\varphi'} : Obj(\mathcal{L} \circ \varphi, \tau) \Rightarrow [L_1]_{\varphi}^{\varphi'}} \qquad \frac{}{\Gamma \vdash \emptyset : (\mathcal{L}, \tau) \Rightarrow \emptyset}$$

$$\frac{\begin{array}{c}\Gamma \vdash L : (\mathcal{L}, \tau) \Rightarrow L_1 \\ \Gamma; x : Obj(\mathcal{L} \circ \varphi, \tau) \vdash a : \mathcal{L}(l) \Rightarrow a_1\end{array}}{\Gamma \vdash L; (l = \varsigma(x, \varphi)a) : (\mathcal{L}, \tau) \Rightarrow L_1; (l = \varsigma(x, \varphi)a_1)}$$

$$\frac{\Gamma \vdash a : Obj(\mathcal{L}, \tau) \Rightarrow a_1}{\Gamma \vdash a : \tau \Rightarrow a_1} \qquad \frac{\Gamma \vdash a : \langle l : \tau; \tau' \rangle \Rightarrow a_1}{\Gamma \vdash a\#l : \tau \Rightarrow a_1\#l}$$

$$\frac{\Gamma \vdash a : Obj(\mathcal{L}, \tau) \Rightarrow a_1}{\Gamma \vdash a.l : \mathcal{L}(l) \Rightarrow a_1.l} \qquad \frac{\begin{array}{c}\Gamma \vdash a : Obj(\mathcal{L}, \tau) \Rightarrow a_1 \\ \Gamma; x : Obj(\mathcal{L}, \tau) \vdash a' : \mathcal{L}(l) \Rightarrow a'_1\end{array}}{\Gamma \vdash (a.l \Leftarrow \varsigma(x)a') : Obj(\mathcal{L}, \tau) \Rightarrow (a_1.l \Leftarrow \varsigma(x)a'_1)}$$

FIG. 2.7: Traduction du calcul de base

$$\frac{\Gamma \vdash L : (\mathcal{L}, \tau) \Rightarrow L_1 \qquad \operatorname{dom}\mathcal{L} = \operatorname{dom} L \cup \operatorname{im}\varphi}{\Gamma \vdash [L]_\varphi : \mathit{Prot}(\mathcal{L} \circ \varphi, \tau, \varphi^{-1}(\operatorname{dom} L)) \Rightarrow [L_1]_\varphi}$$

$$\frac{\Gamma \vdash a : \mathit{Prot}(\mathcal{L}, \tau, \mathcal{D}) \Rightarrow a_1 \qquad \mathcal{D} = \operatorname{dom}\mathcal{L} \qquad \tau = \langle \mathcal{L}|_{\mathcal{P}} \rangle}{\Gamma \vdash \mathtt{new}_{\mathcal{P}}\ a : \tau \Rightarrow \mathtt{new}_{\mathcal{P}}\ a_1} \qquad \frac{\Gamma \vdash a : \mathit{Prot}(\mathcal{L}, \tau, \mathcal{D}) \Rightarrow a_1 \qquad l \notin \operatorname{dom}\mathcal{L}}{\Gamma \vdash (a + l) : \mathit{Prot}((l : \tau'; \mathcal{L}), \tau, \mathcal{D}) \Rightarrow (a_1 + l)}$$

$$\frac{\Gamma \vdash a : \mathit{Prot}(\mathcal{L}, \tau, \mathcal{D}) \Rightarrow a_1 \qquad \Gamma; x : \mathit{Obj}(\mathcal{L}, \tau) \vdash a' : \mathcal{L}(l) \Rightarrow a'_1}{\Gamma \vdash (a.l \Leftarrow \varsigma(x)a') : \mathit{Prot}(\mathcal{L}, \tau, \mathcal{D} \cup \{l\}) \Rightarrow (a_1.l \Leftarrow \varsigma(x)a'_1)}$$

$$\frac{\Gamma \vdash a : \mathit{Prot}(\mathcal{L}, \tau, \mathcal{D}) \Rightarrow a_1 \qquad l \in \mathcal{D}}{\Gamma \vdash (a \setminus l) : \mathit{Prot}(\mathcal{L}|_{\operatorname{dom}\mathcal{L} \setminus \{l\}}, \tau, \mathcal{D} \setminus \{l\}) \Rightarrow (a_1 \setminus l)}$$

$$\frac{\Gamma \vdash a : \mathit{Prot}(\mathcal{L}, \tau, \mathcal{D}) \Rightarrow a_1 \qquad l \in \mathcal{D}}{\Gamma \vdash (a \mathbin{/} l) : \mathit{Prot}(\mathcal{L}, \tau, \mathcal{D} \setminus \{l\}) \Rightarrow (a_1 \mathbin{/} l)}$$

$$\frac{\Gamma \vdash a : \mathit{Prot}(\mathcal{L}, \tau, \mathcal{D}) \Rightarrow a_1 \qquad \mathcal{L}(l) = \mathcal{L}(l')}{\Gamma \vdash (a@(l = l')) : \mathit{Prot}(\mathcal{L}, \tau, \mathcal{D} \cup \{l\}) \Rightarrow (a_1@(l = l'))}$$

$$\frac{\Gamma \vdash a : \mathit{Prot}(\mathcal{L}, \tau, \mathcal{D}) \Rightarrow a_1 \qquad \Gamma \vdash a' : \mathit{Prot}(\mathcal{L}', \tau, \mathcal{D}') \Rightarrow a'_1 \qquad \forall l \in \operatorname{dom}\mathcal{L} \cap \operatorname{dom}\mathcal{L}' \cdot \mathcal{L}(l) = \mathcal{L}'(l)}{\Gamma \vdash (a + a') : \mathit{Prot}((\mathcal{L}; \mathcal{L}'), \tau, \mathcal{D} \cup \mathcal{D}') \Rightarrow (a_1 + a'_1)}$$

FIG. 2.8: Traduction du calcul de base (suite)

$$\frac{\Gamma \vdash a : \mathit{Cl}(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P}) \Rightarrow a_1 \qquad \mathcal{D} = \operatorname{dom}\mathcal{L} \qquad \tau = \langle \mathcal{L}|_{\mathcal{P}} \rangle}{\Gamma \vdash \mathtt{new}^{\mathcal{P}}\ a : \tau \Rightarrow \mathtt{new}_{\mathcal{P}}\ a_1} \qquad \frac{\Gamma, x : \tau \vdash d : \mathit{Cl}(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P}) \Rightarrow a_1}{\Gamma \vdash \mathtt{object}\ (x)\ d\ \mathtt{end} : \mathit{Cl}(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P}) \Rightarrow a_1}$$

$$\frac{\Gamma \vdash a : \theta(\tau') \Rightarrow a'_1 \qquad \vdash \tau' < \tau \Rightarrow a_1}{\Gamma \vdash (a : \tau' < \tau) : \theta(\tau) \Rightarrow a_1(a'_1)}$$

FIG. 2.9: Traduction des expressions

$$\frac{}{\Gamma, x : \tau \vdash \emptyset : Cl(\emptyset, \tau, \emptyset, \emptyset) \Rightarrow [\emptyset]_\emptyset}$$

$$\frac{\Gamma, x : \tau \vdash d : Cl(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P}) \Rightarrow a_1 \qquad l \notin \operatorname{dom} \mathcal{L}}{\Gamma, x : \tau \vdash (d; \texttt{abstract}\ l) : Cl((l : \tau'; \mathcal{L}), \tau, \mathcal{D}, \mathcal{P}) \Rightarrow a_1 + l}$$

$$\frac{\Gamma, x : \tau \vdash d : Cl(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P}) \Rightarrow a_1 \qquad \Gamma; x : Obj(\mathcal{L}, \tau) \vdash a : \mathcal{L}(l) \Rightarrow a'_1}{\Gamma, x : \tau \vdash (d; \texttt{method}\ l = a) : Cl(\mathcal{L}, \tau, \mathcal{D} \cup \{l\}, \mathcal{P}) \Rightarrow a_1.l \Leftarrow \varsigma(x)a'_1}$$

$$\frac{\Gamma, x : \tau \vdash d : Cl(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P}) \Rightarrow a_1 \qquad \mathcal{L}(l) = \mathcal{L}(l')}{\Gamma, x : \tau \vdash (d; \texttt{rename}\ l\ \texttt{as}\ l') : Cl(\mathcal{L}, \tau, \mathcal{D} \cup \{l'\}, \mathcal{P}) \Rightarrow a_1@(l' = l)}$$

$$\frac{\Gamma, x : \tau \vdash d : Cl(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P}) \Rightarrow a_1}{\Gamma, x : \tau \vdash (d; \texttt{public}\ l) : Cl(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P} \cup \{l\}) \Rightarrow a_1}$$

$$\frac{\begin{array}{c}\Gamma, x : \tau \vdash d : Cl(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P}) \Rightarrow a_1 \qquad \Gamma \vdash a : Cl(\mathcal{L}', \tau, \mathcal{D}', \mathcal{P}') \Rightarrow a'_1 \\ \forall l \in \operatorname{dom} \mathcal{L} \cap \operatorname{dom} \mathcal{L}' \cdot \mathcal{L}(l) = \mathcal{L}'(l)\end{array}}{\Gamma, x : \tau \vdash (d; \texttt{inherit}\ a) : Cl((\mathcal{L}; \mathcal{L}'), \tau, \mathcal{D} \cup \mathcal{D}', \mathcal{P} \cup \mathcal{P}') \Rightarrow a_1 + a'_1}$$

FIG. 2.10: Traduction des corps de classes

$$\frac{\vdash \tau'_2 < \tau'_1 \Rightarrow a' \qquad \vdash \tau_1 < \tau_2 \Rightarrow a}{\vdash \tau'_1 \to \tau_1 < \tau'_2 \to \tau_2 \Rightarrow \lambda(x)\lambda(x')a(x(a'(x')))} \qquad \frac{}{\vdash \tau < \tau \Rightarrow \lambda(x)x}$$

$$\frac{\begin{array}{c}\vdash Cl(\mathcal{L}_1, \tau_1, \mathcal{D}_1, \mathcal{P}_1) < Cl(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P}) \Rightarrow a \\ l \in \mathcal{D} \qquad \mathcal{L}' = \mathcal{L}|_{\operatorname{dom} \mathcal{L} \setminus \{l\}} \qquad \mathcal{D}' = \mathcal{D} \setminus \{l\}\end{array}}{\vdash Cl(\mathcal{L}_1, \tau_1, \mathcal{D}_1, \mathcal{P}_1) < Cl(\mathcal{L}', \tau, \mathcal{D}', \mathcal{P}) \Rightarrow \lambda(x)((a(x)) \setminus l)}$$

$$\frac{\vdash Cl(\mathcal{L}_1, \tau_1, \mathcal{D}_1, \mathcal{P}_1) < Cl(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P}) \Rightarrow a \qquad l \in \mathcal{D} \qquad \mathcal{D}' = \mathcal{D} \setminus \{l\}}{\vdash Cl(\mathcal{L}_1, \tau_1, \mathcal{D}_1, \mathcal{P}_1) < Cl(\mathcal{L}, \tau, \mathcal{D}', \mathcal{P}) \Rightarrow \lambda(x)((a(x))\ /\ l)}$$

$$\frac{\vdash Cl(\mathcal{L}_1, \tau_1, \mathcal{D}_1, \mathcal{P}_1) < Cl(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P}) \Rightarrow a \qquad \mathcal{P}' = \mathcal{P} \cup \{l\} \qquad l \notin \mathcal{P}}{\vdash Cl(\mathcal{L}_1, \tau_1, \mathcal{D}_1, \mathcal{P}_1) < Cl(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P}') \Rightarrow a}$$

FIG. 2.11: Traduction des contraintes de type

coercion utilise une série de règles définissant une fonction de traduction (figure 2.11). Cette fonction est appliquée à la traduction de l'expression coercée.

La traduction d'une expression ne doit pas dépendre de son type. Ce choix de sémantique entraîne donc un certain nombre de lourdeurs qui rendraient le langage pénible à utiliser en pratique. Ainsi, on doit indiquer la liste des méthodes publiques lorsque l'on crée un objet, même si cette liste est contenue dans le type de la classe. De même, on doit donner le type d'une classe pour pouvoir modifier son interface par une contrainte. Enfin, une méthode ne peut pas être ajoutée de manière implicite au moment où l'on donne sa définition mais doit être explicitement introduite au préalable. Nous verrons dans le chapitre suivant (plus précisément, en section 3.2 page 85) comment la synthèse de type, et certaines restrictions que nous ferons au système de type afin de la rendre possible, permettent de résoudre tous ces problèmes.

2.3.3 Correction de la traduction

Nous devons d'abord préciser la correspondance entre le type d'une expression avant traduction et son type après traduction. Nous définissons pour cela une relation de traduction des types $\Rightarrow$ par induction sur leur syntaxe. Cette relation remplace les types de classes par des types de prototypes : $\mathit{Cl}(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P}) \Rightarrow \mathit{Prot}(\mathcal{L}', \tau', \mathcal{D})$ si $\mathcal{L} \Rightarrow \mathcal{L}'$ et $\tau \Rightarrow \tau'$. Elle laisse les autres types invariants. Cette relation est étendue aux environnements : $\Gamma \Rightarrow \Gamma'$ si $\operatorname{dom} \Gamma' = \operatorname{dom} \Gamma$ et si pour tout $x \in \operatorname{dom} \Gamma$ on a $\Gamma(x) \Rightarrow \Gamma'(x)$. Nous pouvons maintenant énoncer le théorème de correction :

Théorème 12 (Correction de la traduction) *Si* $\Gamma \vdash a : \tau \Rightarrow a'$, $\Gamma \Rightarrow \Gamma'$, *et* $\tau \Rightarrow \tau'$ *alors* $\Gamma' \vdash a' : \tau'$.

Démonstration. La preuve se fait par induction sur une dérivation de traduction $\Gamma \vdash a : \tau \Rightarrow a'$. Dans le cas des corps de classes d, on montre que si $\Gamma, x : \tau_0 \vdash d : \tau \Rightarrow a$, $\Gamma \Rightarrow \Gamma'$ et $\tau \Rightarrow \tau'$, alors $\Gamma' \vdash a : \mathit{Prot}(\mathcal{L}', \tau, \mathcal{D})$. Enfin, en ce qui concerne la coercion des classes, on montre pour toute substitution θ que si $\vdash \tau' < \tau \Rightarrow a_1$, $\Gamma \Rightarrow \Gamma'$, $\theta(\tau) \Rightarrow \tau_1$ et $\theta(\tau') \Rightarrow \tau_1'$, alors $\Gamma' \vdash a_1 : \tau_1' \to \tau_1$. Nous avons omis les cas concernant la traduction du langages de base, qui sont évidents.

Dans ce qui suit, quand il n'y aura pas d'ambiguïté, nous noterons τ' la traduction du type τ et Γ' la traduction de l'environnement Γ.
Nous commençons par les règles concernant les expressions :

–
$$\frac{\begin{array}{c}\Gamma \vdash a : \mathit{Cl}(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P}) \Rightarrow a_1 \\ \mathcal{D} = \operatorname{dom} \mathcal{L} \qquad \tau = \langle \mathcal{L}|_{\mathcal{P}} \rangle\end{array}}{\Gamma \vdash \mathtt{new}^{\mathcal{P}}\ a : \tau \Rightarrow \mathtt{new}_{\mathcal{P}}\ a_1}$$

La prémisse se traduit en $\Gamma' \vdash a_1 : \mathit{Prot}(\mathcal{L}', \tau', \mathcal{D})$ où $\mathcal{L} \Rightarrow \mathcal{L}'$ et $\tau \Rightarrow \tau'$. On vérifie aisément que l'on a $\mathcal{D} = \operatorname{dom} \mathcal{L}'$ et $\tau' = \langle \mathcal{L}'|_{\mathcal{P}} \rangle$. Ainsi,

$$\frac{\begin{array}{c}\Gamma' \vdash a_1 : \mathit{Prot}(\mathcal{L}', \tau', \mathcal{D}) \\ \mathcal{D} = \operatorname{dom} \mathcal{L}' \qquad \tau' = \langle \mathcal{L}'|_{\mathcal{P}} \rangle\end{array}}{\Gamma' \vdash \mathtt{new}_{\mathcal{P}}\ a_1 : \tau'} \ (\textsc{Constr})$$

–
$$\frac{\Gamma, x : \tau \vdash d : \mathit{Cl}(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P}) \Rightarrow a_1}{\Gamma \vdash \mathtt{object}\ (x)\ d\ \mathtt{end} : \mathit{Cl}(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P}) \Rightarrow a_1}$$

Les deux jugements ont la même traduction.

–

$$\frac{\begin{array}{c}\Gamma \vdash a : \theta(\tau') \Rightarrow a_1' \\ \vdash \tau' < \tau \Rightarrow a_1\end{array}}{\Gamma \vdash (a : \tau' < \tau) : \theta(\tau) \Rightarrow a_1(a_1')}$$

Si l'on suppose que $\theta(\tau) \Rightarrow \tau_1$ et $\theta(\tau') \Rightarrow \tau_1'$, alors la première prémisse se traduit en $\Gamma' \vdash a_1' : \tau_1'$ et la seconde prémisse donne $\Gamma' \vdash a_1 : \tau_1' \to \tau_1$. D'où :

$$\frac{\Gamma' \vdash a_1 : \tau' \to \tau \qquad \Gamma' \vdash a_1' : \tau'}{\Gamma' \vdash a_1(a_1') : \tau}\ (\textsc{App})$$

Considérons maintenant la traduction des corps de classes :

–

$$\frac{}{\Gamma, x : \tau \vdash \emptyset : \mathit{Cl}(\emptyset, \tau, \emptyset, \emptyset) \Rightarrow [\emptyset]_\emptyset}$$

Si $\tau \Rightarrow \tau'$, alors $\mathit{Cl}(\emptyset, \tau, \emptyset, \emptyset)$ se traduit en $\mathit{Prot}(\emptyset, \tau', \emptyset)$, et on a bien :

$$\frac{\dfrac{}{\Gamma \vdash \emptyset : (\emptyset, \tau')}\ (\textsc{Objet-Vide})}{\Gamma \vdash [\emptyset]_\emptyset : \mathit{Prot}(\emptyset, \tau', \emptyset)}\ (\textsc{Prototype})$$

–

$$\frac{\Gamma, x : \tau \vdash d : \mathit{Cl}(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P}) \Rightarrow a_1 \qquad l \notin \mathrm{dom}\,\mathcal{L}}{\Gamma, x : \tau \vdash (d; \texttt{abstract}\ l) : \mathit{Cl}((l : \tau_0; \mathcal{L}), \tau, \mathcal{D}, \mathcal{P}) \Rightarrow a_1 + l}$$

La prémisse se traduit en $\Gamma' \vdash a_1 : \mathit{Prot}(\mathcal{L}', \tau', \mathcal{D}) \Rightarrow a_1$ et donc, si $\tau_0 \Rightarrow \tau_0'$,

$$\frac{\Gamma' \vdash a_1 : \mathit{Prot}(\mathcal{L}', \tau', \mathcal{D}) \qquad l \notin \mathrm{dom}\,\mathcal{L}'}{\Gamma' \vdash (a_1 + l) : \mathit{Prot}((l : \tau_0'; \mathcal{L}'), \tau', \mathcal{D})}\ (\textsc{Ajout})$$

–

$$\frac{\Gamma, x : \tau \vdash d : \mathit{Cl}(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P}) \Rightarrow a_1 \qquad \Gamma; x : \mathit{Obj}(\mathcal{L}, \tau) \vdash a : \mathcal{L}(l) \Rightarrow a_1'}{\Gamma, x : \tau \vdash (d; \texttt{method}\ l = a) : \mathit{Cl}(\mathcal{L}, \tau, \mathcal{D} \cup \{l\}, \mathcal{P}) \Rightarrow a_1.l \Leftarrow \varsigma(x)a_1'}$$

Les prémisses se traduisent en $\Gamma' \vdash a_1 : \mathit{Prot}(\mathcal{L}', \tau', \mathcal{D})$ et en $\Gamma'; x : \mathit{Obj}(\mathcal{L}', \tau') \vdash a_1 : \mathcal{L}'(l)$. D'où :

$$\frac{\begin{array}{c}\Gamma' \vdash a_1 : \mathit{Prot}(\mathcal{L}', \tau', \mathcal{D}) \\ \Gamma'; x : \mathit{Obj}(\mathcal{L}', \tau') \vdash a_1 : \mathcal{L}'(l)\end{array}}{\Gamma' \vdash (a_1.l \Leftarrow \varsigma(x)a_1') : \mathit{Prot}(\mathcal{L}', \tau', \mathcal{D} \cup \{l\})}\ (\textsc{Définition})$$

–

$$\frac{\Gamma, x : \tau \vdash d : \mathit{Cl}(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P}) \Rightarrow a_1 \qquad \mathcal{L}(l) = \mathcal{L}(l')}{\Gamma, x : \tau \vdash (d; \texttt{rename}\ l\ \texttt{as}\ l') : \mathit{Cl}(\mathcal{L}, \tau, \mathcal{D} \cup \{l'\}, \mathcal{P}) \Rightarrow a_1@(l' = l)}$$

La prémisse se traduit en $\Gamma' \vdash a_1 : \mathit{Cl}(\mathcal{L}', \tau', \mathcal{D}, \mathcal{P})$. D'où :

$$\frac{\Gamma' \vdash a_1 : \mathit{Prot}(\mathcal{L}', \tau', \mathcal{D}) \qquad \mathcal{L}'(l) = \mathcal{L}'(l')}{\Gamma' \vdash (a_1@(l = l')) : \mathit{Prot}(\mathcal{L}', \tau', \mathcal{D} \cup \{l\})}\ (\textsc{Copie})$$

–

$$\frac{\Gamma, x : \tau \vdash d : \mathit{Cl}(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P}) \Rightarrow a_1}{\Gamma, x : \tau \vdash (d; \texttt{public}\, l) : \mathit{Cl}(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P} \cup \{l\}) \Rightarrow a_1}$$

Les deux jugements se traduisent en : $\Gamma' \vdash a_1 : \mathit{Cl}(\mathcal{L}', \tau', \mathcal{D}, \mathcal{P})$

–

$$\frac{\Gamma, x : \tau \vdash d : \mathit{Cl}(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P}) \Rightarrow a_1 \qquad \Gamma \vdash a : \mathit{Cl}(\mathcal{L}', \tau, \mathcal{D}', \mathcal{P}') \Rightarrow a'_1 \qquad \forall l \in \operatorname{dom} \mathcal{L} \cap \operatorname{dom} \mathcal{L}' \cdot \mathcal{L}(l) = \mathcal{L}'(l)}{\Gamma, x : \tau \vdash (d; \texttt{inherit}\, a) : \mathit{Cl}((\mathcal{L}; \mathcal{L}'), \tau, \mathcal{D} \cup \mathcal{D}', \mathcal{P} \cup \mathcal{P}') \Rightarrow a_1 + a'_1}$$

Si $\mathit{Cl}(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P}) \Rightarrow \mathit{Prot}(\mathcal{L}_1, \tau_1, \mathcal{D})$ et $\mathit{Cl}(\mathcal{L}', \tau', \mathcal{D}', \mathcal{P}') \Rightarrow \mathit{Prot}(\mathcal{L}'_1, \tau'_1, \mathcal{D}')$, alors les prémisses se traduisent en $\Gamma' \vdash a_1 : \mathit{Prot}(\mathcal{L}, \tau, \mathcal{D})$ et en $\Gamma' \vdash a'_1 : \mathit{Prot}(\mathcal{L}', \tau, \mathcal{D}')$. D'où :

$$\frac{\Gamma' \vdash a_1 : \mathit{Prot}(\mathcal{L}_1, \tau_1, \mathcal{D}) \qquad \Gamma' \vdash a'_1 : \mathit{Prot}(\mathcal{L}'_1, \tau'_1, \mathcal{D}') \qquad \forall l \in \operatorname{dom} \mathcal{L}_1 \cap \operatorname{dom} \mathcal{L}'_1 \cdot \mathcal{L}_1(l) = \mathcal{L}'_1(l)}{\Gamma \vdash (a_1 + a'_1) : \mathit{Prot}((\mathcal{L}_1; \mathcal{L}'_1), \tau_1, \mathcal{D} \cup \mathcal{D}')} \ (\textsc{Fusion})$$

Considérons pour finir les contraintes de classes. Soit θ une substitution quelconque.

–

$$\frac{\vdash \tau'_2 < \tau'_1 \Rightarrow a' \qquad \vdash \tau_1 < \tau_2 \Rightarrow a}{\vdash \tau'_1 \to \tau_1 < \tau'_2 \to \tau_2 \Rightarrow \lambda(x)\lambda(x')a(x(a'(x')))}$$

On vérifie facilement que $\lambda(x)\lambda(x')a(x(a'(x')))$ a le type désiré en utilisant l'hypothèse d'induction et les règles de typage Abs et App.

–

$$\frac{}{\vdash \tau < \tau \Rightarrow \lambda(x)x}$$

On a $\vdash \lambda(x)x : \tau' \to \tau'$ pour tout type τ', et donc en particulier quand $\theta(\tau) \Rightarrow \tau'$.

–

$$\frac{\vdash \mathit{Cl}(\mathcal{L}_1, \tau_1, \mathcal{D}_1, \mathcal{P}_1) < \mathit{Cl}(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P}) \Rightarrow a \qquad l \in \mathcal{D} \qquad \mathcal{L}' = \mathcal{L}|_{\operatorname{dom} \mathcal{L} \setminus \{l\}} \qquad \mathcal{D}' = \mathcal{D} \setminus \{l\}}{\vdash \mathit{Cl}(\mathcal{L}_1, \tau_1, \mathcal{D}_1, \mathcal{P}_1) < \mathit{Cl}(\mathcal{L}', \tau, \mathcal{D}', \mathcal{P}) \Rightarrow \lambda(x)((a(x)) \setminus l)}$$

Supposons $\theta(\mathit{Cl}(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P})) \Rightarrow \mathit{Prot}(\mathcal{L}_2, \tau_2, \mathcal{D})$ et $\theta(\mathit{Cl}(\mathcal{L}', \tau, \mathcal{D}', \mathcal{P})) \Rightarrow \mathit{Prot}(\mathcal{L}'_2, \tau_2, \mathcal{D}')$. Alors il suffit de montrer que pour tout environnement Γ', si $\Gamma' \vdash a(x) : \mathit{Prot}(\mathcal{L}_2, \tau_2, \mathcal{D})$, alors $\Gamma' \vdash (a(x)) \setminus l : \mathit{Prot}(\mathcal{L}'_2, \tau_2, \mathcal{D}')$. Or, on a

$$\frac{\Gamma' \vdash (a(x)) : \mathit{Prot}(\mathcal{L}_2, \tau_2, \mathcal{D}) \qquad l \in \mathcal{D}}{\Gamma' \vdash ((a(x)) \setminus l) : \mathit{Prot}(\mathcal{L}_2|_{\operatorname{dom} \mathcal{L}_2 \setminus \{l\}}, \tau_2, \mathcal{D} \setminus \{l\})} \ (\textsc{Masque})$$

Et on a bien $\mathcal{L}'_2 = \mathcal{L}_2|_{\operatorname{dom} \mathcal{L}_2 \setminus \{l\}}$ et $\mathcal{D}' = \mathcal{D} \setminus \{l\}$

–

$$\frac{\vdash \mathit{Cl}(\mathcal{L}_1, \tau_1, \mathcal{D}_1, \mathcal{P}_1) < \mathit{Cl}(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P}) \Rightarrow a \qquad l \in \mathcal{D} \qquad \mathcal{D}' = \mathcal{D} \setminus \{l\}}{\vdash \mathit{Cl}(\mathcal{L}_1, \tau_1, \mathcal{D}_1, \mathcal{P}_1) < \mathit{Cl}(\mathcal{L}, \tau, \mathcal{D}', \mathcal{P}) \Rightarrow \lambda(x)((a(x)) \mathbin{/} l)}$$

Supposons $\theta(Cl(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P})) \Rightarrow Prot(\mathcal{L}_2, \tau_2, \mathcal{D})$ et $\theta(Cl(\mathcal{L}, \tau, \mathcal{D}', \mathcal{P})) \Rightarrow Prot(\mathcal{L}_2, \tau_2, \mathcal{D}')$. Alors il suffit de montrer que pour tout environnement Γ', si $\Gamma' \vdash a(x) : Prot(\mathcal{L}_2, \tau_2, \mathcal{D})$, alors $\Gamma' \vdash (a(x)) \;/\; l : Prot(\mathcal{L}_2, \tau_2, \mathcal{D}')$. Or, on a

$$\frac{\Gamma' \vdash a(x) : Prot(\mathcal{L}_2, \tau_2, \mathcal{D}) \qquad l \in \mathcal{D}}{\Gamma' \vdash ((a(x)) \;/\; l) : Prot(\mathcal{L}_2, \tau_2, \mathcal{D} \setminus \{l\})} \text{ (OUBLI)}$$

Et on a bien $l \in \mathcal{D}$ et $\mathcal{D}' = \mathcal{D} \setminus \{l\}$.

–

$$\frac{\vdash Cl(\mathcal{L}_1, \tau_1, \mathcal{D}_1, \mathcal{P}_1) < Cl(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P}) \Rightarrow a \qquad \mathcal{P}' = \mathcal{P} \cup \{l\} \qquad l \notin \mathcal{P}}{\vdash Cl(\mathcal{L}_1, \tau_1, \mathcal{D}_1, \mathcal{P}_1) < Cl(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P}') \Rightarrow a}$$

Les deux jugements ont la même interprétation.

■

Chapitre 3

Synthèse de type

Afin que notre extension s'intègre naturellement à ML, il est crucial de pouvoir y adapter l'algorithme de synthèse de type de ML. Nous montrons dans ce chapitre que cela est effectivement possible : nous décrivons un tel algorithme et prouvons sa correction ainsi que sa complétude.

Pour établir ce dernier résultat, nous devons apporter un certain nombre de restrictions aux règles de typage. Ces restrictions ne sont pas très contraignantes en pratique. Dans un deuxième temps, nous montrerons comment ces restrictions, ainsi que la synthèse de type elle-même, permettent de simplifier le langage en rendant certaines opérations implicites. Enfin, comme les types de classes sont assez compliqués, nous proposerons de les simplifier sans perdre trop d'expressivité. Toutes ces simplifications permettent d'obtenir un langage très proche d'Objective Caml.

Finalement, nous décrivons le mécanisme d'abréviations employé dans Objective Caml. Le type d'un objet peut être en effet très large : il contient les types d'un nombre potentiellement important de méthodes, et ces types peuvent eux-mêmes contenir des types d'objets. Il est donc impératif d'avoir un moyen efficace et pratique d'abréger les types. Le mécanisme d'abréviations utilisé généralement dans les compilateurs ML s'avère insuffisant et nous décrirons les améliorations que nous avons dû mettre en œuvre.

3.1 Algorithme de synthèse

Bien que cet algorithme de synthèse pourrait être présenté pour l'intégralité du langage défini dans le chapitre précédent, nous ne considérerons qu'un sous-ensemble de ce langage, qui constitue un langage de surface : il ne nous paraît en effet pas utile de présenter l'algorithme pour une partie du langage qui ne sera pas accessible au programmeur.

3.1.1 Présentation

Nous commençons par préciser le language que nous allons considérer. La syntaxe de ce langage est donnée dans la figure 3.1. Par rapport au langage complet, toutes les constructions concernant les prototypes ont été omises, ainsi

$a ::= x$	Variable
$\mid \lambda(x)a$	Abstraction
$\mid a(a)$	Application
$\mid \texttt{let}\ x = a\ \texttt{in}\ a$	Définition
$\mid (a : \tau :> \tau)$	Contrainte de sous-typage
$\mid a\#l$	Appel de méthode
$\mid a.l$	Appel de méthode interne
$\mid a.l \Leftarrow \varsigma(x)a$	(Re)définition de méthode
$\mid \texttt{new}^{\mathcal{P}}\ a$	Constructeur
$\mid \texttt{object}\ (x)\ d\ \texttt{end}$	Corps de classe
$\mid (a : \tau < \tau)$	Coercion
$d ::= \emptyset$	
$\mid d; \texttt{abstract}\ l$	Introduction de méthode
$\mid d; \texttt{method}\ l = a$	Définition de méthode
$\mid d; \texttt{rename}\ l\ \texttt{as}\ l$	Duplication de méthode
$\mid d; \texttt{public}\ l$	Exportation de méthode
$\mid d; \texttt{inherit}\ a$	Héritage
$\tau ::= \alpha \mid \tau \to \tau \mid \langle\omega\rangle \mid \mu(\alpha)\tau$	Type
$\mid Obj(\mathcal{L}, \tau) \mid Cl(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P})$	
$\omega ::= \rho \mid \emptyset \mid l : \tau; \omega$	Type de rangée
$\sigma ::= \forall(\alpha)\sigma \mid \tau$	Schéma de type

FIG. 3.1: Syntaxe du langage de surface

que les types correspondant. Il n'y a également pas de syntaxe pour définir directement un objet. Le langage est autrement inchangé.

Les règles de typage du langage de surface sont présentées dans la figure 3.2. Nous avons été amenés à apporter un certain nombre de restrictions aux règles de typage données précédemment afin d'obtenir facilement un algorithme de synthèse de type complet.

Ainsi, pour montrer cette propriété, nous avons besoin que le typage soit principal, c'est-à-dire que si $\Gamma \vdash a : \tau$, alors il existe un type τ_0 tel que $\Gamma \vdash a : \tau_0$ et tel que pour tout type τ' tel que $\Gamma \vdash a : \tau'$, τ' est une instance de τ_0. Ce n'est pas le cas avec les règles de typage que nous avons donné précédemment, comme le montre par exemple la fonction $\lambda(x)(\mathtt{new}^{\mathcal{P}}\ x)$. En effet, cette fonction doit avoir un type de la forme $\mathit{Cl}(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P}) \to \tau$ où $\mathcal{L}$ peut définir le type d'un nombre arbitraire de méthodes à la seule condition que ces méthodes soient un sur-ensemble de $\mathcal{P}$. Or des types de classes sont incompatibles si leurs rangées $\mathcal{L}$ n'ont pas le même domaine. Le même problème se pose avec les types d'objets complets $\mathit{Obj}(\mathcal{L}, \tau)$. Nous allons donc restreindre les règles de typage afin qu'elle ne nécessitent pas de « deviner » des types de classes ou des types d'objets complets. Nous définissons pour cela une notion de *types simples* : ce sont les types ne contenant ni type d'objet complet $\mathit{Obj}(\mathcal{L}, \tau)$ ni type de classe $\mathit{Cl}(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P})$. Cette définition est étendue aux substitutions : une substitution θ est *simple* si, pour toute variable de type α, le type $\theta(\alpha)$ est simple. Les règles de typage sont modifiées de manière à ce que tout type devant être « deviné » soit simple (règles VAR-2, ABS-2, CLASSE-COERCION-2 et CLASSE-VIDE-2).

La règle OUBLI pose également un probléme car son application n'est pas *dirigée par la syntaxe*, c'est-à-dire que la syntaxe de l'expression que l'on veut typer ne dicte pas le moment où cette règle doit être appliquée : on a parfois le choix entre l'appliquer ou non. De plus, cette règle ne commute pas toujours avec les autres règles de typage. Ainsi, si le type du corps d'une fonction est $\mathit{Obj}(\mathcal{L}, \tau)$, cette fonction pourra avoir aussi bien le type $\tau' \to \mathit{Obj}(\mathcal{L}, \tau)$ que le type $\tau' \to \tau$, mais ces types sont incompatibles. Nous allons donc rendre déterministe l'application de cette règle et ne l'autoriser qu'à certains endroits. En fait, nous allons l'intégrer aux autres règles de typage. La fonction « Oubli » correspond à une application forcée de la règle OUBLI. Cette fonction est définie par :

$$\begin{array}{rcll} \mathrm{Oubli}(\mathit{Obj}(\mathcal{L}, \tau)) & = & \tau & \\ \mathrm{Oubli}(\tau) & = & \tau & \qquad \text{autrement} \end{array}$$

On l'utilise typiquement dans les règles de typage où deux types venant de prémisses différentes doivent être rendus égaux (règles APP-2, REDÉFINITION-2 et CLASSE-DÉFINITION-2).

Les autres règles de typage sont inchangées. Comme les règles que nous venons de définir sont dérivées des règles de typage du langage complet, les preuves de correction s'appliquent toujours. (Plus précisément, tout programme bien typé avec ces règles est bien typé avec les règles précédentes),

L'algorithme d'unification utilise la fonction d'unification « mgu » définie en section 1.4.4. Il utilise également une fonction dérivée « mguFun », définie par $\mathrm{mguFun}(\tau, \tau', V) = ((\theta_0; \alpha = \alpha), \theta_0(\alpha), V')$, où la variable α est choisie arbitrairement dans $\alpha \in V \setminus (\mathrm{VL}(\tau) \cup \mathrm{VL}(\tau'))$, si $\mathrm{mgu}(\tau, \tau', V \setminus \{\alpha\}) = (\theta_0, V')$. On vérifie aisément que si $(\theta, \tau_0, V') = \mathrm{mguFun}(\tau, \tau', V)$, alors $\theta(\tau) = \theta(\tau') \to$

(Var-2)
$$\frac{(x : \forall(\vec{\alpha})\tau) \in \Gamma \qquad \vec{\tau} \text{ simples}}{\Gamma \vdash x : \tau\{\vec{\tau}/\vec{\alpha}\}}$$

(Abs-2)
$$\frac{\Gamma; x : \tau' \vdash a : \tau \qquad \tau' \text{ simple}}{\Gamma \vdash \lambda(x)a : \tau' \to \tau}$$

(App-2)
$$\frac{\Gamma \vdash a : \mathrm{Oubli}(\tau') \to \tau \qquad \Gamma \vdash a' : \tau'}{\Gamma \vdash a(a') : \tau}$$

(Let)
$$\frac{\Gamma \vdash a : \tau \qquad \Gamma; x : \mathrm{gen}(\tau, \Gamma) \vdash a' : \tau'}{\Gamma \vdash \texttt{let}\ x = a\ \texttt{in}\ a' : \tau'}$$

(Sélection-Privée)
$$\frac{\Gamma \vdash a : Obj(\mathcal{L}, \tau)}{\Gamma \vdash a.l : \mathcal{L}(l)}$$

(Redéfinition-2)
$$\frac{\Gamma \vdash a : Obj(\mathcal{L}, \tau) \quad \Gamma; x : Obj(\mathcal{L}, \tau) \vdash a' : \tau' \qquad \mathrm{Oubli}(\tau') = \mathcal{L}(l)}{\Gamma \vdash (a.l \Leftarrow \varsigma(x)a') : Obj(\mathcal{L}, \tau)}$$

(Classe-Création)
$$\frac{\Gamma \vdash a : Cl(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P}) \quad \mathcal{D} = \mathrm{dom}\,\mathcal{L} \qquad \tau = \langle \mathcal{L}|_{\mathcal{P}} \rangle}{\Gamma \vdash \texttt{new}^{\mathcal{P}}\ a : \tau}$$

(Classe-Corps)
$$\frac{\Gamma, x : \tau \vdash d : Cl(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P})}{\Gamma \vdash \texttt{object}\ (x)\ d\ \texttt{end} : Cl(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P})}$$

(Classe-Coercion-2)
$$\frac{\Gamma \vdash a : \theta(\tau') \quad \vdash \tau' < \tau \qquad \theta \text{ simple}}{\Gamma \vdash (a : \tau' < \tau) : \theta(\tau)}$$

(Classe-Vide-2)
$$\frac{\tau \text{ simple}}{\Gamma, x : \tau \vdash \emptyset : Cl(\emptyset, \tau, \emptyset, \emptyset)}$$

(Classe-Ajout)
$$\frac{\Gamma, x : \tau \vdash d : Cl(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P}) \qquad l \notin \mathrm{dom}\,\mathcal{L}}{\Gamma, x : \tau \vdash (d; \texttt{abstract}\ l) : Cl((l : \tau'; \mathcal{L}), \tau, \mathcal{D}, \mathcal{P})}$$

(Classe-Définition-2)
$$\frac{\Gamma, x : \tau \vdash d : Cl(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P}) \quad \Gamma; x : Obj(\mathcal{L}, \tau) \vdash a : \tau' \qquad \mathrm{Oubli}(\tau') = \mathcal{L}(l)}{\Gamma, x : \tau \vdash (d; \texttt{method}\ l = a) : Cl(\mathcal{L}, \tau, \mathcal{D} \cup \{l\}, \mathcal{P})}$$

(Classe-Renommage)
$$\frac{\Gamma, x : \tau \vdash d : Cl(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P}) \qquad \mathcal{L}(l) = \mathcal{L}(l')}{\Gamma, x : \tau \vdash (d; \texttt{rename}\ l\ \texttt{as}\ l') : Cl(\mathcal{L}, \tau, \mathcal{D} \cup \{l'\}, \mathcal{P})}$$

(Classe-Publique)
$$\frac{\Gamma, x : \tau \vdash d : Cl(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P})}{\Gamma, x : \tau \vdash (d; \texttt{public}\ l) : Cl(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P} \cup \{l\})}$$

(Classe-Héritage)
$$\frac{\Gamma, x : \tau \vdash d : Cl(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P}) \qquad \Gamma \vdash a : Cl(\mathcal{L}', \tau, \mathcal{D}', \mathcal{P}') \quad \forall l \in \mathrm{dom}\,\mathcal{L} \cap \mathrm{dom}\,\mathcal{L}' \cdot \mathcal{L}(l) = \mathcal{L}'(l)}{\Gamma, x : \tau \vdash (d; \texttt{inherit}\ a) : Cl((\mathcal{L}; \mathcal{L}'), \tau, \mathcal{D} \cup \mathcal{D}', \mathcal{P} \cup \mathcal{P}')}$$

Fig. 3.2: Règles de typage du langage de surface

τ_0. Nous supposerons à partir de maintenant que ces fonctions ne sont définies que si la substitution qu'elles retournent est simple. La fonction « mguFun » est utilisée dans l'algorithme de synthèse pour le cas de l'application. Nous ne pouvons pas utiliser la fonction « mgu » dans ce cas car, avec la restriction que nous lui appliquons, elle échouerait dès que le type du résultat de la fonction appliquée n'est pas simple : il ne serait ainsi pas possible de synthétiser le type de l'application d'une classe à l'un de ses arguments.

Nous pouvons maintenant définir l'algorithme de synthèse de type (figures 3.3 et 3.4). Cet algorithme prend en entrée une expression a, un environnement Γ et un ensemble V de variables de types « fraîches ». En cas de succès, il retourne un type τ et une substitution θ tels que $\theta(\Gamma) \vdash a : \tau$, ainsi qu'un sous-ensemble V' de V. Il utilise une fonction Instance qui retourne une instance triviale d'un schéma de type σ : cette fonction vérifie les conditions suivantes :

$$\begin{aligned} \text{Instance}(\tau, V) &= (\tau, V) \\ \text{Instance}(\forall(\alpha)\sigma, V) &= (\sigma\{\alpha'/\alpha\}, V \setminus \{\alpha'\}) \text{ où } \alpha' \in V \end{aligned}$$

L'algorithme de typage des corps de classe est défini simultanément. Il prend en argument un environnement Γ, une variable x, un corps de classe d et un ensemble de variables « fraîches » V et retourne en cas de succès un type τ de la forme $Cl(\mathcal{L}, \tau', \mathcal{D}, \mathcal{P})$, tel que $\theta(\Gamma), x : \tau' \vdash d : \tau$.

L'appel de méthode $\_\#l$ et le sous-typage $(\_ : \tau :> \tau')$ peuvent être considérés comme des valeurs primitives (de types respectifs $\forall(\alpha)\forall(\alpha')(\langle l : \alpha; \alpha'\rangle \to \alpha)$ et $\forall(\vec{\alpha})(\tau \to \tau')$, où les variables $\vec{\tau}$ sont les variables libres des types τ et τ'). Il n'y a donc pas besoin de cas spécifiques pour ces deux constructions.

3.1.2 Correction de l'algorithme

La preuve de correction repose sur le lemme suivant :

Lemme 13 (Stabilité par substitution) *Si $\Gamma \vdash a : \tau$ alors pour toute substitution simple θ, $\theta(\Gamma) \vdash a : \theta(\tau)$.*

Démonstration. La preuve est quasiment la même que celle du lemme 4 page 44. Elle ne pose pas de difficulté particulière. Le fait que la substitution θ soit simple assure que pour tout type τ, $\theta(\text{Oubli}(\tau)) = \text{Oubli}(\theta(\tau))$. Cette égalité est utilisée à plusieurs reprise dans la preuve, notamment dans le cas de la règle App-2. ∎

Nous pouvons maintenant aborder le théorème de correction.

Théorème 14 (Correction de l'algorithme de synthèse de type) *Si $(\tau, \theta, V') = \text{Infère}(\Gamma, a, V)$ alors $\theta(\Gamma) \vdash a : \tau$.*

Démonstration. La preuve se fait par induction sur la syntaxe de l'expression a. Nous montrons simultanément le même résultat sur les corps de classes, c'est-à-dire que si $\text{InfèreCorps}(\Gamma, x, d, V) = (Cl(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P}), \theta, V')$, alors $\theta(\Gamma), x : \tau \vdash d : Cl(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P})$. Pour établir la preuve, nous montrons aussi simultanément que θ est simple.

- $a = x$

  On a $(\tau, V') = \text{Instance}(\Gamma(x), V)$ et θ est l'identité.

Infère(Γ, a, V) est le triplet (τ, θ, V') défini par :

- Si $a = x$:
  Alors $(\tau, V') = \text{Instance}(\Gamma(x), V)$ et θ est l'identité.
- Si $a = \lambda(x)a_1$:
  Soit $\alpha \in V$.
  Soit $(\tau_1, \theta_1, V_1) = \text{Infère}((\Gamma; x : \alpha), a_1, V \setminus \{\alpha\})$.
  Alors $\tau = \theta_1(\alpha) \to \tau_1$, $V' = V_1$ et $\theta = \theta_1$.
- Si $a = a_1(a'_1)$:
  Soit $(\tau_1, \theta_1, V_1) = \text{Infère}(\Gamma, a_1, V)$.
  Soit $(\tau_2, \theta_2, V_2) = \text{Infère}(\theta_1(\Gamma), a'_1, V_1)$.
  Soit $(\theta_3, \tau_3, V_3) = \text{mguFun}(\theta_2(\tau_1), \text{Oubli}(\tau_2), V_2 \setminus \{\alpha\})$
  Alors $\tau = \tau_3$, $V' = V_3$ et $\theta = \theta_3 \circ \theta_2 \circ \theta_1$.
- Si $a = \texttt{let } x = a'_1 \texttt{ in } a_1$:
  Soit $(\tau_1, \theta_1, V_1) = \text{Infère}(\Gamma, a'_1, V)$.
  Soit $(\tau_2, \theta_2, V_2) = \text{Infère}((\theta_1(\Gamma); x : \text{gen}(\tau_1, \theta_1(\Gamma))), a_1, V_1)$.
  Alors $\tau = \tau_2$, $V' = V_2$ et $\theta = \theta_2 \circ \theta_1$.
- Si $a = a_1.l$:
  Soit $(Obj(\mathcal{L}_1, \tau_1), \theta_1, V_1) = \text{Infère}(\Gamma, a_1, V)$.
  Alors $\tau = \mathcal{L}_1(l)$, $V' = V_1$ et $\theta = \theta_1$.
- Si $a = a_1.l \Leftarrow \varsigma(x)a'_1$:
  Soit $(Obj(\mathcal{L}_1, \tau_1), \theta_1, V_1) = \text{Infère}(\Gamma, a_1, V)$.
  Soit $(\tau_2, \theta_2, V_2) = \text{Infère}((\theta_1(\Gamma); x : Obj(\mathcal{L}_1, \tau_1)), a'_1, V_1)$.
  Soit $(\theta_3, V_3) = \text{mgu}(\theta_2(\mathcal{L}_1(l)), \text{Oubli}(\tau_2), V_2)$.
  Alors $\tau = (\theta_3 \circ \theta_2)(Obj(\mathcal{L}_1, \tau_1))$, $V' = V_3$ et $\theta = \theta_3 \circ \theta_2 \circ \theta_1$.
- Si $a = \texttt{new}^{\mathcal{P}_0}\ a_1$:
  Soit $(Cl(\mathcal{L}_1, \tau_1, \mathcal{D}_1, \mathcal{P}_1), \theta_1, V_1) = \text{Infère}(\Gamma, a_1, V)$.
  On doit avoir $\mathcal{P}_0 = \mathcal{P}_1$ et $\mathcal{D}_1 = \text{dom}\,\mathcal{L}_1$.
  Soit $(\theta_2, V_2) = \text{mgu}(\tau_1, \langle \mathcal{L}_1|_{\mathcal{P}_1} \rangle, V_1)$.
  Alors $\tau = \theta_2(\tau_1)$, $V' = V_2$ et $\theta = \theta_2 \circ \theta_1$.
- Si $a = \texttt{object}\ (x)\ d\ \texttt{end}$:
  Soit $(\tau_1, \theta_1, V_1) = \text{InfèreCorps}(\Gamma, x, d, V)$.
  Alors $\tau = \tau_1$, $V = V_1$ et $\theta = \theta_1$
- Si $a = (a_1 : \tau_0 < \tau'_0)$:
  On doit avoir $\vdash \tau_0 < \tau'_0$.
  Soit $(\tau_1, \theta_1, V_1) = \text{Infère}(\Gamma, a'_1, V)$.
  Quitte à renommer les variables libres de τ_0 et τ'_0, on suppose que ces variables sont dans V_1. Soit $(\theta_2, V_2) = \text{mgu}(\tau_1, \tau_0, V_1)$.
  Alors $\tau = \theta_2(\tau'_0)$, $V' = V_2$ et $\theta = \theta_2 \circ \theta_1$.

FIG. 3.3: Algorithme de synthèse de type (expressions)

InfèreCorps(Γ, x, d, V) est le triplet (τ, θ, V') défini par :

- Si $d = \emptyset$:
  Soit $\alpha \in V$.
  Alors $\tau = Cl(\emptyset, \alpha, \emptyset, \emptyset)$, θ est l'identité et $V' = V \setminus \{\alpha\}$
- Si $d = d_1; \texttt{abstract}\ l$:
  Soit $(Cl(\mathcal{L}_1, \tau_1, \mathcal{D}_1, \mathcal{P}_1), \theta_1, V_1) = \text{InfèreCorps}(\Gamma, x, d_1, V)$.
  On doit avoir $l \notin \operatorname{dom} \mathcal{L}_1$.
  Soit $\alpha \in V$.
  Alors $\tau = Cl((l : \alpha; \mathcal{L}_1), \tau_1, \mathcal{D}_1, \mathcal{P}_1)$, $\theta = \theta_1$ et $V' = V_1 \setminus \{\alpha\}$.
- Si $d = d_1; \texttt{method}\ l = a$:
  Soit $(Cl(\mathcal{L}_1, \tau_1, \mathcal{D}_1, \mathcal{P}_1), \theta_1, V_1) = \text{InfèreCorps}(\Gamma, x, d_1, V)$.
  Soit $(\tau_2, \theta_2, V_2) = \text{Infère}((\theta_1(\Gamma); x : Obj(\mathcal{L}_1, \tau_1)), a, V_1)$.
  Soit $(\theta_3, V_3) = \text{mgu}(\text{Oubli}(\tau_2), \theta_2(\mathcal{L}_1(l)), V_2)$.
  Alors $\tau = (\theta_3 \circ \theta_2)(Cl(\mathcal{L}_1, \tau_1, \mathcal{D}_1 \cup \{l\}, \mathcal{P}_1))$, $\theta = \theta_3 \circ \theta_2 \circ \theta_1$ et $V' = V_3$.
- Si $d = d_1; \texttt{rename}\ l\ \texttt{as}\ l'$:
  Soit $(Cl(\mathcal{L}_1, \tau_1, \mathcal{D}_1, \mathcal{P}_1), \theta_1, V_1) = \text{InfèreCorps}(\Gamma, x, d_1, V)$.
  Soit $(\theta_2, V_2) = \text{mgu}(\mathcal{L}_1(l), \mathcal{L}_1(l'), V_1)$.
  Alors $\tau = \theta_2(Cl(\mathcal{L}_1, \tau_1, \mathcal{D}_1 \cup \{l'\}, \mathcal{P}_1))$, $\theta = \theta_2 \circ \theta_1$ et $V' = V_2$.
- Si $d = d_1; \texttt{public}\ l$:
  Soit $(Cl(\mathcal{L}_1, \tau_1, \mathcal{D}_1, \mathcal{P}_1), \theta_1, V_1) = \text{InfèreCorps}(\Gamma, x, d_1, V)$.
  Alors $\tau = Cl(\mathcal{L}_1, \tau_1, \mathcal{D}_1, \mathcal{P}_1 \cup \{l\})$, $\theta = \theta_1$ et $V' = V_1$.
- Si $d = d_1; \texttt{inherit}\ a$:
  Soit $(Cl(\mathcal{L}_1, \tau_1, \mathcal{D}_1, \mathcal{P}_1), \theta_1, V_1) = \text{InfèreCorps}(\Gamma, x, d_1, V)$.
  Soit $(Cl(\mathcal{L}_2, \tau_2, \mathcal{D}_2, \mathcal{P}_2), \theta_2, V_2) = \text{Infère}(\theta_1(\Gamma), a, V_1)$.
  Soit $(\theta_3, V_3) = \text{mgu}((\theta_2(\mathcal{L}_1); \mathcal{L}_2), (\mathcal{L}_2; \theta_2(\mathcal{L}_1)), V_2)$.
  Soit $(\theta_4, V_4) = \text{mgu}((\theta_3 \circ \theta_2)(\tau_1), \theta_3(\tau_2), V_3)$.
  Alors $\tau = (\theta_4 \circ \theta_3)(Cl((\theta_2(\mathcal{L}_1); \mathcal{L}_2), \tau_2, \mathcal{D}_1 \cup \mathcal{D}_2, \mathcal{P}_1 \cup \mathcal{P}_2))$, $\theta = \theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1$ et $V' = V_4$.

FIG. 3.4: Algorithme de synthèse de type (classes)

Par définition de l'instanciation, si $\Gamma(x) = \forall(\vec{\alpha})\tau_0$, alors τ est de la forme $\tau_0\{\vec{\tau}/\vec{\alpha}\}$, où les types $\vec{\tau}$ sont en fait des variables, et sont donc simples. La règle VAR-2 permet alors d'établir le jugement voulu.

$$\frac{(x : \forall(\vec{\alpha})\tau_0) \in \Gamma \qquad \vec{\tau} \text{ simples}}{\Gamma \vdash x : \tau_0\{\vec{\tau}/\vec{\alpha}\}} \ (\textsc{Var-2})$$

D'autre part, la substitution identité est bien simple.

- $a = \lambda(x)a_1$
  On a : $(\tau_1, \theta_1, V_1) = \text{Infère}((\Gamma; x : \alpha), a_1, V \setminus \{\alpha\})$. En appliquant l'hypothèse de récurrence à a_1, il vient donc : $\theta_1(\Gamma; x : \alpha) \vdash a_1 : \tau_1$.
  D'autre part $\tau = \theta_1(\alpha) \to \tau_1$, $V' = V_1$ et $\theta = \theta_1$.
  Comme θ est simple, la règle ABS-2 permet donc de conclure :

$$\frac{\theta(\Gamma); x : \theta(\alpha) \vdash a_1 : \tau_1 \qquad \theta(\alpha) \text{ simple}}{\theta(\Gamma) \vdash \lambda(x)a_1 : \theta(\alpha) \to \tau_1} \ (\textsc{Abs-2})$$

- $a = a_1(a'_1)$
  L'hypothèse de récurrence nous donne :

$$\theta_1(\Gamma) \vdash a_1 : \tau_1$$

  et

$$(\theta_2 \circ \theta_1)(\Gamma) \vdash a'_1 : \tau_2$$

  On a $\theta = \theta_3 \circ \theta_2 \circ \theta_1$. D'où, comme θ_2 et θ_3 sont simples, en utilisant le lemme 13,

$$\theta(\Gamma) \vdash a_1 : (\theta_3 \circ \theta_2)(\tau_1)$$

  et

$$\theta(\Gamma) \vdash a'_1 : \theta_3(\tau_2)$$

  Or, par définition de θ_3, $\theta_3(\theta_2(\tau_1)) = \theta_3(\text{Oubli}(\tau_2)) \to \tau_3$.
  Donc, finalement,

$$\frac{\theta(\Gamma) \vdash a_1 : \text{Oubli}(\theta_3(\tau_2)) \to \tau_3 \qquad \theta(\Gamma) \vdash a'_1 : \theta_3(\tau_2)}{\theta(\Gamma) \vdash a_1(a'_1) : \tau_3} \ (\textsc{App-2})$$

  D'autre part, θ est bien simple, car c'est la composée de substitutions simples.

- $a = \texttt{let}\ x = a'_1\ \texttt{in}\ a_1$
  L'hypothèse de récurrence nous donne :

$$\theta_1(\Gamma) \vdash a'_1 : \tau_1$$

  et

$$\theta_2(\theta_1(\Gamma); x : \text{gen}(\tau_1, \theta_1(\Gamma))) \vdash a_1 : \tau_2$$

  Soit ψ_2 une substitution simple telle que $\psi_2(\theta_1(\Gamma)) = \theta_2(\theta_1(\Gamma)) = \theta(\Gamma)$ et telle que les images des variables généralisables de τ_1 par ψ_2 soient des variables distinctes non liées dans $\psi_2(\theta_1(\Gamma))$.
  On vérifie que $\theta_2(\text{gen}(\tau_1, \theta_1(\Gamma))) = \text{gen}(\psi_2(\tau_1), \psi_2(\theta_1(\Gamma)))$.

Il vient alors (en utilisant le lemme 13 pour la première prémisse) :

$$\frac{\theta(\Gamma) \vdash a'_1 : \psi_2(\tau_1) \qquad \theta(\Gamma); x : \mathrm{gen}(\psi_2(\tau_1), \theta(\Gamma)) \vdash a_1 : \tau_2}{\theta(\Gamma) \vdash \texttt{let } x = a'_1 \texttt{ in } a_1 : \tau_2} \ (\textsc{Let})$$

comme désiré.

D'autre part, comme θ_1 et θ_2 sont simples, leur composée θ l'est aussi.

- $a = a_1.l$

  L'hypothèse d'induction nous donne :

$$\theta_1(\Gamma) \vdash a_1 : Obj(\mathcal{L}_1, \tau_1)$$

  et θ_1 est simple.

  La règle de typage suivante permet alors de conclure :

$$\frac{\theta(\Gamma) \vdash a_1 : Obj(\mathcal{L}_1, \tau_1)}{\theta(\Gamma) \vdash a_1.l : \mathcal{L}_1(l)} \ (\textsc{Sélection-Privée})$$

- $a = a_1.l \Leftarrow \varsigma(x)a'_1$

  L'hypothèse d'induction nous donne

$$\theta_1(\Gamma) \vdash a_1 : Obj(\mathcal{L}_1, \tau_1)$$

  et

$$\theta_2(\theta_1(\Gamma); x : Obj(\mathcal{L}_1, \tau_1)) \vdash a'_1 : \tau_2$$

  Comme $\theta = \theta_3 \circ \theta_2 \circ \theta_1$,

$$\theta(\Gamma) \vdash a_1 : (\theta_3 \circ \theta_2)(Obj(\mathcal{L}_1, \tau_1))$$

  et

$$\theta(\Gamma); x : (\theta_3 \circ \theta_2)(Obj(\mathcal{L}_1, \tau_1)) \vdash a'_1 : \theta_3(\tau_2)$$

  Par définition de θ_3, on a $(\theta_3 \circ \theta_2)(\mathcal{L}_1(l)) = \theta_3(\mathrm{Oubli}(\tau_2))$

  La règle de typage suivante nous permet alors d'établir le jugement voulu :

$$\frac{\begin{array}{c}\theta(\Gamma) \vdash a : (\theta_3 \circ \theta_2)(Obj(\mathcal{L}_1, \tau_1)) \\ \theta(\Gamma); x : (\theta_3 \circ \theta_2)(Obj(\mathcal{L}_1, \tau_1)) \vdash a' : \theta_3(\tau_2) \\ (\theta_3 \circ \theta_2)(\mathcal{L}_1(l)) = \mathrm{Oubli}(\theta_3(\tau_2))\end{array}}{\theta(\Gamma) \vdash (a.l \Leftarrow \varsigma(x)a') : (\theta_3 \circ \theta_2)(Obj(\mathcal{L}_1, \tau_1))} \ (\textsc{Redéfinition-2})$$

  D'autre part, la substitution θ est simple, car obtenue par composition de substitution simples.

- $a = \texttt{new}^{\mathcal{P}_1}\ a_1$

  L'hypothèse d'induction nous donne :

$$\theta_1(\Gamma) \vdash a_1 : Cl(\mathcal{L}_1, \tau_1, \mathcal{D}_1, \mathcal{P}_1)$$

  D'où :

$$\theta(\Gamma) \vdash a_1 : \theta_2(Cl(\mathcal{L}_1, \tau_1, \mathcal{D}_1, \mathcal{P}_1))$$

  D'autre part, on sait que $\mathcal{D}_1 = \mathrm{dom}\,\mathcal{L}_1$, et donc que $\mathcal{D}_1 = \mathrm{dom}(\theta_2 \circ \mathcal{L}_1)$.

Enfin, par définition de θ_2, $\theta_2(\tau_1) = \theta_2(\langle \mathcal{L}_1|_{\mathcal{P}_1}\rangle)$. On peut donc conclure en utilisant la règle de typage suivante :

$$\frac{\begin{array}{c}\theta(\Gamma) \vdash a_1 : \theta_2(\mathit{Cl}(\mathcal{L}_1, \tau_1, \mathcal{D}_1, \mathcal{P}_1)) \\ \mathcal{D}_1 = \mathrm{dom}(\theta_2 \circ \mathcal{L}_1) \qquad \theta_2(\tau_1) = \theta_2(\langle \mathcal{L}_1|_{\mathcal{P}_1}\rangle)\end{array}}{\theta(\Gamma) \vdash \mathtt{new}^{\mathcal{P}_1}\ a_1 : \theta_2(\tau_1)} \ (\textsc{Classe-Création})$$

- $a = \mathtt{object}\ (x)\ d\ \mathtt{end}$
  Clair
- $a = (a_1 : \tau_0 < \tau_0')$
  L'hypothèse de récurrence nous donne :

  $$\theta_1(\Gamma) \vdash a_1 : \tau_1$$

  D'où, comme $\theta = \theta_2 \circ \theta_1$,

  $$\theta(\Gamma) \vdash a_1 : \theta_2(\tau_1)$$

  Par définition de θ_2, on a : $\theta_2(\tau_1) = \theta_2(\tau_0)$.
  D'autre part, on a $\vdash \tau_0 < \tau_0'$.
  D'où

  $$\frac{\begin{array}{c}\theta(\Gamma) \vdash a_1 : \theta_2(\tau_0) \\ \vdash \tau_0 < \tau_0' \qquad \theta_2 \text{ simple}\end{array}}{\theta(\Gamma) \vdash (a_1 : \tau_0 < \tau_0') : \theta_2(\tau_0')} \ (\textsc{Classe-Coercion-2})$$

  Enfin, comme θ_1 et θ_2 sont simples, θ l'est aussi.
- $d = \emptyset$
  Clair.
- $d = d_1; \mathtt{abstract}\ l$
  L'hypothèse de récurrence nous donne :

  $$\theta_1(\Gamma), x : \tau_1 \vdash d_1 : \mathit{Cl}(\mathcal{L}_1, \tau_1, \mathcal{D}_1, \mathcal{P}_1)$$

  où $\theta_1 (= \theta)$ est simple.
  La règle de typage suivante permet d'établir le jugement recherché :

  $$\frac{\theta(\Gamma), x : \tau_1 \vdash d : \mathit{Cl}(\mathcal{L}_1, \tau_1, \mathcal{D}_1, \mathcal{P}_1) \qquad l \notin \mathrm{dom}\,\mathcal{L}_1}{\theta(\Gamma), x : \tau_1 \vdash (d; \mathtt{abstract}\ l) : \mathit{Cl}((l : \alpha; \mathcal{L}_1), \tau_1, \mathcal{D}_1, \mathcal{P}_1)} \ (\textsc{Classe-Ajout})$$

- $d = d_1; \mathtt{method}\ l = a$
  L'hypothèse de récurrence nous donne :

  $$\theta_1(\Gamma), x : \tau_1 \vdash d_1 : \mathit{Cl}(\mathcal{L}_1, \tau_1, \mathcal{D}_1, \mathcal{P}_1)$$

  et

  $$\theta_2(\theta_1(\Gamma); x : \mathit{Obj}(\mathcal{L}_1, \tau_1)) \vdash a : \tau_2$$

  On a $\theta = \theta_3 \circ \theta_2 \circ \theta_1$. D'où :

  $$\theta(\Gamma), x : (\theta_3 \circ \theta_2)(\tau_1) \vdash d_1 : (\theta_3 \circ \theta_2)(\mathit{Cl}(\mathcal{L}_1, \tau_1, \mathcal{D}_1, \mathcal{P}_1))$$

et

$$\theta(\Gamma); x : (\theta_3 \circ \theta_2)(Obj(\mathcal{L}_1, \tau_1)) \vdash a : \theta_3(\tau_2)$$

Or, par définition de θ_3, $\theta_3(\text{Oubli}(\tau_2)) = \theta_3(\theta_2(\mathcal{L}_1(l)))$.
Donc, finalement, en utilisant la règle CLASSE-DÉFINITION-2,

$$\frac{\begin{array}{c}\theta(\Gamma), x : (\theta_3 \circ \theta_2)(\tau_1) \vdash d : (\theta_3 \circ \theta_2)(Cl(\mathcal{L}_1, \tau_1, \mathcal{D}_1, \mathcal{P}_1)) \\ \theta(\Gamma); x : (\theta_3 \circ \theta_2)(Obj(\mathcal{L}, \tau_1)) \vdash a : \theta_3(\tau_2) \\ \text{Oubli}(\theta_3(\tau_2)) = (\theta_3 \circ \theta_2)(\mathcal{L}_1(l))\end{array}}{\theta(\Gamma), x : (\theta_3 \circ \theta_2)(\tau_1) \vdash (d; \texttt{method}\, l = a) : (\theta_3 \circ \theta_2)(Cl(\mathcal{L}_1, \tau_1, \mathcal{D}_1 \cup \{l\}, \mathcal{P}_1))}$$

D'autre part, θ est clairement simple.

- $d = d_1; \texttt{rename}\, l \;\texttt{as}\; l'$
- $d = d_1; \texttt{public}\, l$

  Ces deux cas sont similaires au cas $d = d_1; \texttt{abstract}\, l$.
- $d = d_1; \texttt{inherit}\, a$

  L'hypothèse de récurrence nous donne :

$$\theta_1(\Gamma), x : \tau_1 \vdash d_1 : Cl(\mathcal{L}_1, \tau_1, \mathcal{D}_1, \mathcal{P}_1)$$

et

$$\theta_2(\theta_1(\Gamma)) \vdash a : Cl(\mathcal{L}_2, \tau_2, \mathcal{D}_2, \mathcal{P}_2)$$

On a $\theta = \theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1$. D'où :

$$\theta(\Gamma), x : (\theta_4 \circ \theta_3)(\tau_2) \vdash d : (\theta_4 \circ \theta_3 \circ \theta_2)(Cl(\mathcal{L}_1, \tau_1, \mathcal{D}_1, \mathcal{P}_1))$$

et

$$\theta(\Gamma) \vdash a : (\theta_4 \circ \theta_3)(Cl(\mathcal{L}_2, \tau_2, \mathcal{D}_2, \mathcal{P}_2))$$

D'autre part, posons $\mathcal{L} = (\theta_4 \circ \theta_3 \circ \theta_2)(\mathcal{L}_1)$ et $\mathcal{L}' = (\theta_4 \circ \theta_3)(\mathcal{L}_2)$. On a $\theta_3(\theta_2(\mathcal{L}_1); \mathcal{L}_2) = \theta_3(\mathcal{L}_2; \theta_2(\mathcal{L}_1))$ et donc $(\mathcal{L}; \mathcal{L}') = (\mathcal{L}'; \mathcal{L})$. On en déduit aisément que $\forall l \in \operatorname{dom} \mathcal{L} \cap \operatorname{dom} \mathcal{L}' \cdot \mathcal{L}(l) = \mathcal{L}'(l)$.
On obtient donc par la règle CLASSE-HÉRITAGE, comme désiré,

$$\frac{\begin{array}{c}\theta(\Gamma), x : (\theta_4 \circ \theta_3)(\tau_2) \vdash d : (\theta_4 \circ \theta_3 \circ \theta_2)(Cl(\mathcal{L}_1, \tau_1, \mathcal{D}_1, \mathcal{P}_1)) \\ \theta(\Gamma) \vdash a : (\theta_4 \circ \theta_3)(Cl(\mathcal{L}_2, \tau_2, \mathcal{D}_2, \mathcal{P}_2)) \\ \forall l \in \operatorname{dom} \mathcal{L} \cap \operatorname{dom} \mathcal{L}' \cdot \mathcal{L}(l) = \mathcal{L}'(l)\end{array}}{\begin{array}{c}\theta(\Gamma), x : (\theta_4 \circ \theta_3)(\tau_2) \vdash (d; \texttt{inherit}\, a) : \\ (\theta_4 \circ \theta_3)(Cl((\theta_2(\mathcal{L}_1); \mathcal{L}_2), \tau_2, \mathcal{D}_1 \cup \mathcal{D}_2, \mathcal{P}_1 \cup \mathcal{P}_2))\end{array}}$$

∎

3.1.3 Complétude de l'algorithme

Si V est un ensemble de variables de types et θ et θ' sont des substitutions, on dira que $\theta = \theta'$ hors de V si pour toute variable $\alpha \notin V$, $\theta(\alpha) = \theta'(\alpha)$.

Lemme 15 (Renforcement de l'environnement) *Si* $\Gamma; x : \sigma \vdash a : \tau$ *et* $\sigma = \sigma'\{\vec{\tau}/\vec{\alpha}\}$ *alors* $\Gamma; x : \forall(\alpha)\sigma' \vdash a : \tau$.

Théorème 16 (Complétude de l'algorithme de synthèse de type) *Soit Γ un environnement, V un ensemble infini de variables de types, θ une substitution simple, a une expression et τ un type. On suppose que $V \cap \mathrm{VL}(\Gamma) = \emptyset$ et que $\theta(\Gamma) \vdash a : \tau$. Alors il existe un type τ', des substitutions θ' et θ'' et un ensemble de variables V' tels que θ'' soit simple,* $\mathrm{Infère}(\Gamma, a, V) = (\tau', \theta', V')$*, $\tau = \theta''(\tau')$ et $\theta = \theta'' \circ \theta'$ hors de V.*

Les conditions $V \cap \mathrm{VL}(\Gamma) = \emptyset$ et $\theta = \theta'' \circ \theta'$ hors de V signifient que les substitutions θ et $\theta'' \circ \theta'$ se comportent de la même manière sur le problème initial. On a donc $(\theta'' \circ \theta')(\Gamma) = \theta(\Gamma)$. Le jugement initial peut donc aussi s'écrire $(\theta'' \circ \theta')(\Gamma) \vdash a : \theta''(\tau')$: c'est une instance du jugement synthétisé $\theta'(\Gamma) \vdash a : \tau'$.

Démonstration. On commence par remarquer que si $V \cap \mathrm{VL}(\Gamma) = \emptyset$ et $\mathrm{Infère}(\Gamma, a, V) = (\tau', \theta', V')$, alors $\mathrm{VL}(\tau') \cap V' = \emptyset$, les variables de V' sont hors de portée[1] de θ' et $V' \subseteq V$. En particulier, on a $V' \cap \mathrm{VL}(\theta'(\Gamma)) = \emptyset$.

La preuve se fait alors par induction sur la syntaxe de l'expression a.

- $a = x$

  Comme $\theta(\Gamma) \vdash x : \tau$, on $x \in \mathrm{dom}\,\Gamma$. $\mathrm{Infère}(\Gamma, x, V)$ est donc bien définie et vaut (τ', θ', V'), où θ' est l'identité et $(\tau', V') = \mathrm{Instance}(\Gamma(x), V)$.

  Comme τ est une instance de $\theta(\Gamma(x)) = (\theta(\Gamma))(x)$ et $\mathrm{VL}(\Gamma(x)) \cap V = \emptyset$, il existe une substitution simple θ'' telle que $\tau = \theta''(\tau')$ et $\theta = \theta'' = \theta'' \circ \theta'$ hors de V.

  En effet, $\Gamma(x)$ est de la forme $\forall(\vec{\alpha})\tau_0$. On peut supposer que les variables $\vec{\alpha}$ sont hors de portée de θ et appartiennent à V. Ainsi, $\theta(\Gamma(x)) = \forall(\vec{\alpha})\theta(\tau_0)$. Alors, $\tau = \theta_1(\theta(\tau_0))$ pour une certaine substitution simple θ_1 ne modifiant que les variables $\vec{\alpha}$. D'autre part, $\tau' = \theta_2(\tau_0)$, où θ_2 est la substitution associant aux variables $\vec{\alpha}$ des variables distinctes $\vec{\beta}$ appartenant à V. Alors $\tau = (\theta_1 \circ \theta \circ \theta_2^{-1})(\tau')$ où θ_2^{-1} est la substitution associant aux variables $\vec{\beta}$ les variables $\vec{\alpha}$ (comme $\mathrm{VL}(\Gamma(x)) \cap V = \emptyset$, on a bien $\theta_2^{-1}(\tau') = \tau_0$. Finalement, si $\alpha \notin V$, comme $\theta_2^{-1}(\alpha) = \alpha$ et $\mathrm{VL}(\theta(\alpha))$ ne contient aucune des variables $\vec{\alpha}$, $(\theta_1 \circ \theta \circ \theta_2^{-1})(\alpha) = \theta(\alpha)$. La substitution (simple car composée de substitutions simples) $\theta'' = \theta_1 \circ \theta \circ \theta_2^{-1}$ convient donc bien.

- $a = \lambda(x)a_1$

  On a la dérivation de typage suivante :

  $$\frac{\theta(\Gamma); x : \tau_0' \vdash a_1 : \tau_0 \qquad \tau_0' \text{ simple}}{\theta(\Gamma) \vdash \lambda(x)a_1 : \tau_0' \to \tau_0}\ (\textsc{Abs-2})$$

  avec $\tau = \tau_0$.

  Soient $\alpha \in V$ et $V_0 = V \setminus \{\alpha\}$.

  On a $\mathrm{VL}((\Gamma; x : \alpha)) \cap (V \setminus \{\alpha\}) = \emptyset$. De plus, si θ_0 est la substitution (simple) qui à α associe τ_0' et qui à toute autre variable α' associe $\theta(\alpha')$, alors $\theta_0(\Gamma; x : \alpha) \vdash a_1 : \tau_0$. On peut donc appliquer l'hypothèse de récurrence à a_1, ce qui nous donne $(\tau_1, \theta_1, V_1) = \mathrm{Infère}((\Gamma; x : \alpha), a_1, V \setminus \{\alpha\})$ avec $\tau_0 = \theta''(\tau_1)$ et $\theta_0 = \theta'' \circ \theta_1$ hors de V_0, pour une certaine substitution simple θ''.

[1] On dit qu'une variable α est hors de portée d'une substitution θ si $\theta(\alpha) = \alpha$ et si pour toute variable de type α', si $\alpha \in \mathrm{VL}(\theta(\alpha'))$ alors $\alpha = \alpha'$

En posant $\tau' = \theta_1(\alpha) \to \tau_1$, $V' = V_1$ et $\theta' = \theta_1$, on a donc $\text{Infère}(\Gamma, a, V) = (\tau', \theta', V')$.
De plus,

$$\begin{aligned} \theta''(\tau') &= \theta''(\theta_1(\alpha)) \to \theta''(\tau_1) \\ &= \theta_0(\alpha) \to \tau_0 \\ &= \tau_0' \to \tau_0 \\ &= \tau \end{aligned}$$

Enfin, comme $\theta_0 = \theta$ hors de V et $\theta_0 = \theta'' \circ \theta'$ hors de V_0, on a $\theta = \theta'' \circ \theta'$ hors de V.
D'où le résultat.

– $a = a_1(a_1')$
On a la dérivation de typage suivante :

$$\frac{\theta(\Gamma) \vdash a_1 : \text{Oubli}(\tau_0') \to \tau \qquad \theta(\Gamma) \vdash a_1' : \tau_0'}{\theta(\Gamma) \vdash a_1(a_1') : \tau} \ (\text{App-2})$$

L'hypothèse de récurrence appliquée à a_1 nous donne

$$(\tau_1, \theta_1, V_1) = \text{Infère}(\Gamma, a_1, V)$$

avec $\text{Oubli}(\tau_0') \to \tau = \theta_1''(\tau_1)$ et $\theta = \theta_1'' \circ \theta_1$ hors de V, pour une certaine substitution simple θ_1''.
On a en particulier $\theta_1''(\theta_1(\Gamma)) \vdash a_1' : \tau_0'$ et, d'après la remarque faite en début de preuve, $V_1 \cap \text{VL}(\theta_1(\Gamma)) = \emptyset$.
On peut donc appliquer l'hypothèse de récurrence à a_1', ce qui nous donne $(\tau_2, \theta_2, V_2) = \text{Infère}(\theta_1(\Gamma), a_1', V_1)$ avec $\tau_0' = \theta_2''(\tau_2)$ et $\theta_1'' = \theta_2'' \circ \theta_2$ hors de V_1, pour une certaine substitution simple θ_2''.
Soit $\alpha \in V_2$ et θ_0 la substitution qui associe τ à α et $\theta_2''(\alpha')$ à toute autre variable α'.
Alors $\text{Oubli}(\tau_0') \to \tau = \theta_1''(\tau_1) = \theta_2''(\theta_2(\tau_1)) = \theta_0(\theta_2(\tau_1))$ (car α est hors de portée de θ_2 et $\alpha \notin \text{VL}(\tau_1)$).
D'autre part, $\theta_0(\text{Oubli}(\tau_2) \to \alpha) = \theta_2''(\text{Oubli}(\tau_2)) \to \tau = \text{Oubli}(\tau_0') \to \tau$ (car $\alpha \notin \text{VL}(\tau_2)$). Les types $\theta_2(\tau_1)$ et $\text{Oubli}(\tau_2) \to \alpha$ sont donc unifiables : il existe θ_3 et V_3 tels que $(\theta_3, V_3) = \text{mgu}(\theta_2(\tau_1), \text{Oubli}(\tau_2) \to \alpha, V_2 \setminus \{\alpha\})$. De plus, il existe une substitution θ'' telle que $\theta_0 = \theta'' \circ \theta_3$.
On peut supposer que cette substitution est simple. En effet, si $\alpha' \neq \alpha$, alors $(\theta'' \circ \theta_3)(\alpha') = \theta_2''(\alpha')$ et $\theta_2''(\alpha')$ est simple. D'autre part, $(\theta'' \circ \theta_3)(\alpha) = \tau$. Si τ est simple, on peut conclure. Supposons donc que τ n'est pas simple. Il suffit de montrer que la restriction de θ'' aux variables libres de $\theta_3(\alpha)$ est simple. Comme la restriction de θ_0 aux variables libres de τ_1 est simple, elle n'introduit pas de nouveaux types de classes ou types d'objets complets à τ_1. Ce type doit donc avoir la forme $\tau_3' \to \tau_3$ où τ_3 n'est pas simple. Alors, $(\theta'' \circ \theta_3)(\tau_3) = \tau = \theta_1''(\tau_3)$. Comme θ_1'' est simple, la restriction de θ'' aux variables libres de $\theta_3(\tau_3) = \theta_3(\alpha)$ l'est aussi, ce que l'on voulait montrer.
Posons alors $\tau' = \theta_3(\alpha)$, $V' = V_3$, $\theta' = \theta_3 \circ \theta_2 \circ \theta_1$. On a $\text{Infère}(\Gamma, a, V) = (\tau', \theta', V')$.

De plus,

$$\begin{array}{rcl} \theta''(\tau') & = & \theta''(\theta_3(\alpha)) \\ & = & \theta_0(\alpha) \\ & = & \tau \end{array}$$

Il ne reste plus qu'à montrer que $\theta = \theta'' \circ \theta'$ hors de V. On a $\theta'' \circ \theta' = \theta'' \circ \theta_3 \circ \theta_2 \circ \theta_1 = \theta_0 \circ \theta_2 \circ \theta_1$. Soit $\alpha' \notin V$. D'après la remarque faite en début de preuve, $\alpha \notin \mathrm{VL}(\theta_1(\alpha'))$ puis $\alpha \notin \mathrm{VL}((\theta_2 \circ \theta_1)(\alpha'))$. De plus, les variables de V_1 sont hors de portée de θ_1 et donc $V_1 \cap \mathrm{VL}(\theta_1(\alpha')) = \emptyset$. D'où :

$$\begin{array}{rcl} (\theta'' \circ \theta')(\alpha') & = & (\theta_0 \circ \theta_2 \circ \theta_1)(\alpha') \\ & = & (\theta_2'' \circ \theta_2 \circ \theta_1)(\alpha') \\ & = & (\theta_1'' \circ \theta_1)(\alpha') \\ & = & \theta(\alpha') \end{array}$$

D'où le résultat.

– $a = \texttt{let}\ x = a_1'\ \texttt{in}\ a_1$

On a la dérivation de typage suivante :

$$\frac{\theta(\Gamma) \vdash a_1' : \tau_0 \qquad \theta(\Gamma); x : \mathrm{gen}(\tau_0, \theta(\Gamma)) \vdash a_1 : \tau}{\theta(\Gamma) \vdash \texttt{let}\ x = a_1'\ \texttt{in}\ a_1 : \tau}\ (\textsc{Let})$$

L'hypothèse de récurrence appliquée à a_1' nous donne

$$(\tau_1, \theta_1, V_1) = \mathrm{Infère}(\Gamma, a_1', V)$$

avec $\tau_0 = \theta_1''(\tau_1)$ et $\theta = \theta_1'' \circ \theta_1$ hors de V, pour une certaine substitution simple θ_1''.

On a en particulier $\theta(\Gamma) = (\theta_1'' \circ \theta_1)(\Gamma)$. On vérifie facilement que le schéma de type $\theta_1''(\mathrm{gen}(\tau_1, \theta_1(\Gamma)))$ est plus général que le schéma de type $\mathrm{gen}(\theta_1''(\tau_1), \theta_1''(\theta_1(\Gamma))) = \mathrm{gen}(\tau_0, \theta(\Gamma))$. Par conséquent, le lemme 15 nous donne :

$$\theta(\Gamma); x : \theta_1''(\mathrm{gen}(\tau_1, \theta_1(\Gamma))) \vdash a_1 : \tau$$

c'est-à-dire

$$\theta_1''(\theta_1(\Gamma); x : \mathrm{gen}(\tau_1, \theta_1(\Gamma))) \vdash a_1 : \tau$$

De plus, $V_1 \cap \mathrm{VL}(\theta_1(\Gamma); x : \mathrm{gen}(\tau_1, \theta_1(\Gamma))) = V_1 \cap \mathrm{VL}(\theta_1(\Gamma)) = \emptyset$ (d'après la remarque faite en début de preuve).

On peut donc appliquer l'hypothèse de récurrence à a_1, ce qui nous donne $(\tau_2, \theta_2, V_2) = \mathrm{Infère}((\theta_1(\Gamma); x : \mathrm{gen}(\tau_1, \theta_1(\Gamma))), a_1, V_1)$ avec $\tau = \theta_2''(\tau_2)$ et $\theta_1'' = \theta_2'' \circ \theta_2$ hors de V_1, pour une certaine substitution simple θ_2''.

Posons $\tau' = \tau_2$, $V' = V_2$, $\theta' = \theta_2 \circ \theta_1$ et $\theta'' = \theta_2''$. On a $\mathrm{Infère}(\Gamma, a, V) = (\tau', \theta', V')$ et θ'' est simple.

De plus,

$$\begin{array}{rcl} \theta''(\tau') & = & \theta_2''(\tau_2) \\ & = & \tau \end{array}$$

Il ne reste plus qu'à montrer que $\theta = \theta'' \circ \theta'$ hors de V. On a $\theta'' \circ \theta' = \theta''_2 \circ \theta_2 \circ \theta_1$. Comme les variables de V_1 sont hors de portée de θ_1 et $\theta''_2 \circ \theta_2 = \theta''_1$ hors de V_1, on a alors $\theta'' \circ \theta' = \theta''_1 \circ \theta_1$ hors de V. D'où, finalement, $\theta = \theta'' \circ \theta'$ hors de V comme désiré.

- $a = a_1.l$

  On a la dérivation de typage suivante :

$$\frac{\theta(\Gamma) \vdash a_1 : Obj(\mathcal{L}, \tau_0)}{\theta(\Gamma) \vdash a_1.l : \mathcal{L}(l)} \text{ (Sélection-Privée)}$$

  avec $\tau = \mathcal{L}(l)$.

  L'hypothèse de récurrence appliquée à a_1 nous donne

$$(\tau'_1, \theta_1, V_1) = \text{Infère}(\Gamma, a_1, V)$$

  avec $Obj(\mathcal{L}, \tau_0) = \theta''_1(\tau'_1)$ et $\theta = \theta''_1 \circ \theta_1$ hors de V, pour une certaine substitution simple θ''_1.

  Comme θ''_1 est simple, τ'_1 s'écrit $Obj(\mathcal{L}_1, \tau_1)$.

  Posons $\tau = \mathcal{L}_1(l)$, $V' = V_1$ et $\theta = \theta_1$.

  On vérifie aisément que ces valeurs conviennent.

- $a = a_1.l \Leftarrow \varsigma(x)a'_1$

  On a la dérivation de typage suivante :

$$\frac{\begin{array}{c}\theta(\Gamma) \vdash a_1 : Obj(\mathcal{L}, \tau_0) \\ \theta(\Gamma); x : Obj(\mathcal{L}, \tau_0) \vdash a'_1 : \tau'_0 \qquad \text{Oubli}(\tau'_0) = \mathcal{L}(l)\end{array}}{\theta(\Gamma) \vdash (a_1.l \Leftarrow \varsigma(x)a'_1) : Obj(\mathcal{L}, \tau_0)} \text{ (Redéfinition-2)}$$

  avec $\tau = Obj(\mathcal{L}, \tau_0)$.

  L'hypothèse de récurrence appliquée à a_1 nous donne

$$(\tau'_1, \theta_1, V_1) = \text{Infère}(\Gamma, a_1, V)$$

  avec $Obj(\mathcal{L}, \tau_0) = \theta''_1(\tau'_1)$ et $\theta = \theta''_1 \circ \theta_1$ hors de V, pour une certaine substitution simple θ''_1.

  Comme θ''_1 est simple, τ'_1 s'écrit $Obj(\mathcal{L}_1, \tau_1)$.

  On a alors $\theta''_1(\theta_1(\Gamma); x : Obj(\mathcal{L}_1, \tau_1)) \vdash a'_1 : \tau'_0$ et, d'après la remarque faite en début de preuve, $V_1 \cap \text{VL}(\theta_1(\Gamma); x : Obj(\mathcal{L}_1, \tau_1)) = V_1 \cap (\text{VL}(\theta_1(\Gamma)) \cup \text{VL}(Obj(\mathcal{L}_1, \tau_1))) = \emptyset$.

  On peut donc appliquer l'hypothèse de récurrence à a'_1, ce qui nous donne $(\tau_2, \theta_2, V_2) = \text{Infère}((\theta_1(\Gamma); x : Obj(\mathcal{L}_1, \tau_1)), a'_1, V_1)$ avec $\tau'_0 = \theta''_2(\tau_2)$ et $\theta''_1 = \theta''_2 \circ \theta_2$ hors de V_1, pour une certaine substitution simple θ''_2.

  On a alors $\theta''_2(\theta_2(\mathcal{L}_1(l))) = \theta''_1(\mathcal{L}_1(l)) = \mathcal{L}(l)$ et $\theta''_2(\text{Oubli}(\tau_2)) = \text{Oubli}(\tau'_0) = \mathcal{L}(l)$. Les types $\theta_2(\mathcal{L}_1(l))$ et $\text{Oubli}(\tau_2)$ sont donc unifiables : il existe θ_3 et V_3 tels que $(\theta_3, V_3) = \text{mgu}(\theta_2(\mathcal{L}_1(l)), \text{Oubli}(\tau_2), V_2)$. De plus, il existe une substitution θ'' telle que $\theta''_2 = \theta'' \circ \theta_3$. Comme θ''_2 est simple, on peut choisir θ'' simple.

  Posons alors $\tau' = (\theta_3 \circ \theta_2)(Obj(\mathcal{L}_1, \tau_1))$, $V' = V_3$ et $\theta' = \theta_3 \circ \theta_2 \circ \theta_1$.

  On a $\text{Infère}(\Gamma, a, V) = (\tau', \theta', V')$.

De plus,

$$\begin{aligned}
\theta''(\tau') &= \theta''((\theta_3 \circ \theta_2)(Obj(\mathcal{L}_1, \tau_1))) \\
&= (\theta''_2 \circ \theta_2)(Obj(\mathcal{L}_1, \tau_1)) \\
&= (\theta''_1)(Obj(\mathcal{L}_1, \tau_1)) \\
&= Obj(\mathcal{L}, \tau_0) = \tau
\end{aligned}$$

Il ne reste plus qu'à montrer que $\theta = \theta'' \circ \theta'$ hors de V. On a $\theta'' \circ \theta' = \theta'' \circ \theta_3 \circ \theta_2 \circ \theta_1 = \theta''_2 \circ \theta_2 \circ \theta_1$. Soit $\alpha' \notin V$. D'après la remarque faite en début de preuve, les variables de V_1 sont hors de portée de θ_1 et donc $V_1 \cap \mathrm{VL}(\theta_1(\alpha')) = \emptyset$. D'où :

$$\begin{aligned}
(\theta'' \circ \theta')(\alpha') &= (\theta''_2 \circ \theta_2 \circ \theta_1)(\alpha') \\
&= (\theta''_1 \circ \theta_1)(\alpha') \\
&= \theta(\alpha')
\end{aligned}$$

D'où le résultat.

- $a = \mathtt{new}^{\mathcal{P}_0}\ a_1$
  On a la dérivation de typage suivante :

$$\frac{\theta(\Gamma) \vdash a_1 : Cl(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P}_0) \qquad \mathcal{D} = \mathrm{dom}\,\mathcal{L} \qquad \tau = \langle \mathcal{L}|_{\mathcal{P}_0} \rangle}{\theta(\Gamma) \vdash \mathtt{new}^{\mathcal{P}_0}\ a_1 : \tau} \ (\textsc{Classe-Création})$$

  L'hypothèse de récurrence appliquée à a_1 nous donne

$$(\tau'_1, \theta_1, V_1) = \mathrm{Infère}(\Gamma, a_1, V)$$

  avec $Cl(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P}_0) = \theta''_1(\tau'_1)$ et $\theta = \theta''_1 \circ \theta_1$ hors de V, pour une certaine substitution simple θ''_1.
  Comme θ''_1 est simple, τ'_1 s'écrit $Cl(\mathcal{L}_1, \tau_1, \mathcal{D}, \mathcal{P}_0)$. De plus, comme $\mathrm{dom}\,\mathcal{L}_1 = \mathrm{dom}\,\mathcal{L}$, on a bien $\mathrm{dom}\,\mathcal{L}_1 = \mathcal{D}$.
  D'autre part, $\theta''_1(\tau_1) = \tau = \langle \mathcal{L}|_{\mathcal{P}_0} \rangle = \theta''_1(\langle \mathcal{L}_1|_{\mathcal{P}_0} \rangle)$. Les types τ_1 et $\langle \mathcal{L}_1|_{\mathcal{P}_0} \rangle$ sont donc unifiables : il existe θ_2 et V_2 tels que $(\theta_2, V_2) = \mathrm{mgu}(\tau_1, \langle \mathcal{L}_1|_{\mathcal{P}_0} \rangle, V_1)$.
  De plus, il existe une substitution θ'' telle que $\theta''_1 = \theta'' \circ \theta_2$. Comme θ''_1 est simple, on peut choisir θ'' simple.
  Posons $\tau' = \theta_2(\tau_1)$, $V' = V_2$ et $\theta' = \theta_2 \circ \theta_1$.
  On a $\mathrm{Infère}(\Gamma, a, V) = (\tau', \theta', V')$.
  De plus,

$$\begin{aligned}
\theta''(\tau') &= \theta''(\theta_2(\tau_1)) \\
&= \theta''_1(\tau_1) \\
&= \tau
\end{aligned}$$

  Finalement, $\theta'' \circ \theta' = \theta$ hors de V. En effet, on a $\theta'' \circ \theta' = \theta'' \circ \theta_2 \circ \theta_1 = \theta''_1 \circ \theta_1$ et, d'autre part, $\theta''_1 \circ \theta_1 = \theta$ hors de V.
  D'où le résultat.
- $a = \mathtt{object}\ (x)\ d\ \mathtt{end}$
  Clair

- $a = (a_1 : \tau_0 < \tau'_0)$

  On a la dérivation suivante :

  $$\frac{\begin{array}{c}\theta(\Gamma) \vdash a_1 : \theta_0(\tau_0) \\ \vdash \tau_0 < \tau'_0 \qquad \theta_0 \text{ simple}\end{array}}{\theta(\Gamma) \vdash (a_1 : \tau_0 < \tau'_0) : \theta_0(\tau'_0)}\ (\textsc{Classe-Coercion-2})$$

  avec $\tau = \theta_0(\tau'_0)$.

  L'hypothèse de récurrence appliquée à a_1 nous donne

  $$(\tau_1, \theta_1, V_1) = \text{Infère}(\Gamma, a_1, V)$$

  avec $\theta_0(\tau_0) = \theta''_1(\tau'_1)$ et $\theta = \theta''_1 \circ \theta_1$ hors de V, pour une certaine substitution simple θ''_1.

  On peut supposer que les variables libres de τ_0 et τ'_0 soient dans V_1. En particulier, elles sont distinctes de celles de τ'_1. Il existe alors une substitution simple θ''_0 telle que $\theta''_0(\tau_0) = \theta_0(\tau_0)$, $\theta''_0(\tau'_0) = \theta_0(\tau'_0)$ et $\theta''_0 = \theta''_1$ hors de V_1. Les types τ_0 et τ'_1 sont alors unifiables : il existe θ_2 et V_2 tels que $(\theta_2, V_2) = \text{mgu}(\tau_1, \tau_0, V_1)$. De plus, il existe une substitution θ'' telle que $\theta''_0 = \theta'' \circ \theta_2$. Comme θ''_0 est simple, on peut choisir θ'' simple.

  Posons alors $\tau' = \theta_2(\tau'_0)$, $V' = V_2$ et $\theta' = \theta_2 \circ \theta_1$.

  On a $\text{Infère}(\Gamma, a, V) = (\tau', \theta', V')$.

  De plus,

  $$\begin{array}{rcl}\theta''(\tau') & = & \theta''(\theta_2(\tau'_0)) \\ & = & \theta''_0(\tau'_0) \\ & = & \theta_0(\tau'_0) \\ & = & \tau\end{array}$$

  Il ne reste plus qu'à montrer que $\theta = \theta'' \circ \theta'$ hors de V.

  En effet, $\theta'' \circ \theta' = \theta'' \circ \theta_2 \circ \theta_1 = \theta''_0 \circ \theta_1$. De plus, les variables de V_1 sont hors de portées de θ_1 et $\theta''_0 = \theta''_1$ hors de V_1. Par conséquent, $\theta''_0 \circ \theta_1 = \theta''_1 \circ \theta_1$ hors de V_1. Et, finalement, $\theta'' \circ \theta' = \theta''_1 \circ \theta_1 = \theta$ hors de V.

- $d = \emptyset$

  On a la dérivation suivante :

  $$\frac{\tau_0 \text{ simple}}{\theta(\Gamma), x : \tau_0 \vdash \emptyset : Cl(\emptyset, \tau_0, \emptyset, \emptyset)}\ (\textsc{Classe-Vide-2})$$

  avec $\tau = Cl(\emptyset, \tau_0, \emptyset, \emptyset)$.

  Soit $\alpha \in V$. Soit θ' l'identité. Soit θ'' la substitution qui associe τ_0 à α et qui est égale à θ ailleurs.

  Posons $\tau' = Cl(\emptyset, \alpha, \emptyset, \emptyset)$, et $V' = V \setminus \{\alpha\}$.

  On a $\text{InfèreCorps}(\Gamma, x, d, V) = (\tau', \theta', V')$.

  De plus, $\theta''(\tau') = \tau$ et $\theta = \theta'' \circ \theta'$ hors de V.

  D'où le résultat.

- $d = d_1; \texttt{abstract}\ l$

  Clair

- $d = d_1; \texttt{method}\, l = a$

  On a la dérivation de typage suivante (règle CLASSE-DÉFINITION-2) :

$$\frac{\theta(\Gamma), x : \tau_0 \vdash d_1 : Cl(\mathcal{L}, \tau_0, \mathcal{D}, \mathcal{P}) \qquad \theta(\Gamma); x : Obj(\mathcal{L}, \tau_0) \vdash a : \tau_0' \qquad \text{Oubli}(\tau_0') = \mathcal{L}(l)}{\theta(\Gamma), x : \tau_0 \vdash (d_1; \texttt{method}\, l = a) : Cl(\mathcal{L}, \tau_0, \mathcal{D} \cup \{l\}, \mathcal{P})}$$

  avec $\tau = Cl(\mathcal{L}, \tau_0, \mathcal{D} \cup \{l\}, \mathcal{P})$.

  L'hypothèse de récurrence appliquée à d_1 nous donne

$$(\tau_1', \theta_1, V_1) = \text{InfèreCorps}(\Gamma, x, d_1, V)$$

  avec $\tau_1' = Cl(\mathcal{L}_1, \tau_1, \mathcal{D}_1, \mathcal{P}_1)$, $Cl(\mathcal{L}, \tau_0, \mathcal{D}, \mathcal{P}) = \theta_1''(\tau_1')$ et $\theta = \theta_1'' \circ \theta_1$ hors de V, pour une certaine substitution simple θ_1''.

  On a en particulier $\theta_1''(\theta_1(\Gamma); x : Obj(\mathcal{L}_1, \tau_1)) \vdash a : \tau_0'$ et, d'après la remarque faite en début de preuve, $V_1 \cap \text{VL}(\theta_1(\Gamma); x : Obj(\mathcal{L}_1, \tau_1)) = \emptyset$.

  On peut donc appliquer l'hypothèse de récurrence à a, ce qui nous donne

$$(\tau_2, \theta_2, V_2) = \text{Infère}(\theta_1(\Gamma); x : Obj(\mathcal{L}_1, \tau_1), a, V_1)$$

  avec $\tau_0' = \theta_2''(\tau_2)$ et $\theta_1'' = \theta_2'' \circ \theta_2$ hors de V_1, pour une certaine substitution simple θ_2''.

  On a alors $\theta_2''(\theta_2(\mathcal{L}_1(l))) = \theta_1''(\mathcal{L}_1(l)) = \mathcal{L}(l)$ et $\theta_2''(\text{Oubli}(\tau_2)) = \text{Oubli}(\tau_0') = \mathcal{L}(l)$. Les types $\theta_2(\mathcal{L}_1(l))$ et $\text{Oubli}(\tau_2)$ sont donc unifiables : il existe θ_3 et V_3 tels que $(\theta_3, V_3) = \text{mgu}(\text{Oubli}(\tau_2), \theta_2(\mathcal{L}_1(l)), V_2)$.

  De plus, il existe une substitution θ'' telle que $\theta_2'' = \theta'' \circ \theta_3$. Comme θ_2'' est simple, on peut choisir θ'' simple.

  Posons alors $\tau' = (\theta_3 \circ \theta_2)(Cl(\mathcal{L}_1, \tau_1, \mathcal{D}_1 \cup \{l\}, \mathcal{P}_1))$, $V' = V_3$ et $\theta' = \theta_3 \circ \theta_2 \circ \theta_1$.

  On a $\text{InfèreCorps}(\Gamma, x, d, V) = (\tau', \theta', V')$.

  De plus,

$$\begin{array}{rcl} \theta''(\tau') & = & \theta''((\theta_3 \circ \theta_2)(Cl(\mathcal{L}_1, \tau_1, \mathcal{D}_1 \cup \{l\}, \mathcal{P}_1))) \\ & = & (\theta_2'' \circ \theta_2)(Cl(\mathcal{L}_1, \tau_1, \mathcal{D}_1 \cup \{l\}, \mathcal{P}_1)) \\ & = & \theta_1''(Cl(\mathcal{L}_1, \tau_1, \mathcal{D}_1 \cup \{l\}, \mathcal{P}_1)) \\ & = & Cl(\mathcal{L}, \tau_0, \mathcal{D} \cup \{l\}, \mathcal{P}) = \tau \end{array}$$

  Il ne reste plus qu'à montrer que $\theta = \theta'' \circ \theta'$ hors de V. On a $\theta'' \circ \theta' = \theta'' \circ \theta_3 \circ \theta_2 \circ \theta_1 = \theta_2'' \circ \theta_2 \circ \theta_1$. Soit $\alpha' \notin V$. D'après la remarque faite en début de preuve, les variables de V_1 sont hors de portée de θ_1 et donc $V_1 \cap \text{VL}(\theta_1(\alpha')) = \emptyset$. D'où :

$$\begin{array}{rcl} (\theta'' \circ \theta')(\alpha') & = & (\theta_2'' \circ \theta_2 \circ \theta_1)(\alpha') \\ & = & (\theta_1'' \circ \theta_1)(\alpha') \\ & = & \theta(\alpha') \end{array}$$

  D'où le résultat.

- $d = d_1; \texttt{rename}\, l\, \texttt{as}\, l'$

  Clair

- $d = d_1; \texttt{public}\, l$
  Clair
- $d = d_1; \texttt{inherit}\, a$
  On a la dérivation de typage suivante :

$$\frac{\begin{array}{c}\theta(\Gamma), x : \tau_0 \vdash d_1 : Cl(\mathcal{L}, \tau_0, \mathcal{D}, \mathcal{P}) \\ \theta(\Gamma) \vdash a : Cl(\mathcal{L}', \tau_0, \mathcal{D}', \mathcal{P}') \\ \forall l \in \operatorname{dom} \mathcal{L} \cap \operatorname{dom} \mathcal{L}' \cdot \mathcal{L}(l) = \mathcal{L}'(l)\end{array}}{\begin{array}{c}\theta(\Gamma), x : \tau_0 \vdash (d_1; \texttt{inherit}\, a) : \\ Cl((\mathcal{L}; \mathcal{L}'), \tau_0, \mathcal{D} \cup \mathcal{D}', \mathcal{P} \cup \mathcal{P}')\end{array}} \text{ (Classe-Héritage)}$$

et $\tau = Cl((\mathcal{L}; \mathcal{L}'), \tau_0, \mathcal{D} \cup \mathcal{D}', \mathcal{P} \cup \mathcal{P}')$.
L'hypothèse de récurrence appliquée à d_1 nous donne

$$(\tau_1', \theta_1, V_1) = \text{InfèreCorps}(\Gamma, x, d_1, V)$$

avec $\tau_1' = Cl(\mathcal{L}_1, \tau_1, \mathcal{D}_1, \mathcal{P}_1)$, $Cl(\mathcal{L}, \tau_0, \mathcal{D}, \mathcal{P}) = \theta_1''(\tau_1')$ et $\theta = \theta_1'' \circ \theta_1$ hors de V, pour une certaine substitution simple θ_1''.
On a en particulier $\theta_1''(\theta_1(\Gamma)) \vdash a : Cl(\mathcal{L}', \tau_0, \mathcal{D}', \mathcal{P}')$ et, d'après la remarque faite en début de preuve, $V_1 \cap \mathrm{VL}(\theta_1(\Gamma)) = \emptyset$.
On peut donc appliquer l'hypothèse de récurrence à a, ce qui nous donne

$$((Cl(\mathcal{L}_2, \tau_2, \mathcal{D}_2, \mathcal{P}_2), \theta_2, V_2), \theta_2, V_2) = \text{Infère}(\theta_1(\Gamma), a, V_1)$$

avec $Cl(\mathcal{L}', \tau_0, \mathcal{D}', \mathcal{P}') = \theta_2''((Cl(\mathcal{L}_2, \tau_2, \mathcal{D}_2, \mathcal{P}_2), \theta_2, V_2))$ et $\theta_1'' = \theta_2'' \circ \theta_2$ hors de V_1, pour une certaine substitution simple θ_2''.
On a alors $\theta_2''(\theta_2(\mathcal{L}_1)) = \theta_1''(\mathcal{L}_1) = \mathcal{L}(l)$ et $\theta_2''(\mathcal{L}_2) = \mathcal{L}'$. Comme $\forall l \in \operatorname{dom} \mathcal{L} \cap \operatorname{dom} \mathcal{L}'$, on a $(\mathcal{L}; \mathcal{L}') = (\mathcal{L}'; mty)$ et donc $\theta_2''((\theta_2(\mathcal{L}_1); \mathcal{L}_2)) = \theta_2''((\mathcal{L}_2; \theta_2(\mathcal{L}_1)))$ Les types $(\theta_2(\mathcal{L}_1); \mathcal{L}_2)$ et $(\mathcal{L}_2; \theta_2(\mathcal{L}_1))$ sont donc unifiables : il existe θ_3 et V_3 tels que $(\theta_3, V_3) = \operatorname{mgu}((\theta_2(\mathcal{L}_1); \mathcal{L}_2), (\mathcal{L}_2; \theta_2(\mathcal{L}_1)), V_2)$
De plus, il existe une substitution θ_3'' telle que $\theta_2'' = \theta_3'' \circ \theta_3$. Comme θ_2'' est simple, on peut choisir θ_3'' simple.
Alors, $\theta_3''((\theta_3 \circ \theta_2)(\tau_1)) = \theta_2''(\theta_2(\tau_1)) = \theta_1''(\tau_1) = \tau_0$ et $\theta_3''(\theta_3(\tau_2)) = \theta_2''(\tau_2) = \tau_0$. Les types $(\theta_3 \circ \theta_2)(\tau_1)$ et $\theta_3(\tau_2)$ sont donc unifiables : il existe θ_4 et V_4 tels que $(\theta_4, V_4) = \operatorname{mgu}((\theta_3 \circ \theta_2)(\tau_1), \theta_3(\tau_2), V_3)$.
De plus, il existe une substitution θ'' telle que $\theta_3'' = \theta'' \circ \theta_4$. Comme θ_3'' est simple, on peut choisir θ'' simple.
Posons $\tau' = (\theta_4 \circ \theta_3)(Cl((\theta_2(\mathcal{L}_1); \mathcal{L}_2), \tau_2, \mathcal{D}_1 \cup \mathcal{D}_2, \mathcal{P}_1 \cup \mathcal{P}_2))$, $\theta' = \theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1$ et $V' = V_4$.
On a $\text{InfèreCorps}(\Gamma, x, d, V) = (\tau', \theta', V')$.
De plus,

$$\begin{array}{rcl}\theta''(\tau') & = & \theta''((\theta_4 \circ \theta_3)(Cl((\theta_2(\mathcal{L}_1); \mathcal{L}_2), \tau_2, \mathcal{D}_1 \cup \mathcal{D}_2, \mathcal{P}_1 \cup \mathcal{P}_2))) \\ & = & \theta_2''(Cl((\theta_2(\mathcal{L}_1); \mathcal{L}_2), \tau_2, \mathcal{D}_1 \cup \mathcal{D}_2, \mathcal{P}_1 \cup \mathcal{P}_2)) \\ & = & Cl((\mathcal{L}; \mathcal{L}'), \tau_0, \mathcal{D} \cup \mathcal{D}', \mathcal{P} \cup \mathcal{P}') = \tau\end{array}$$

Il ne reste plus qu'à montrer que $\theta = \theta'' \circ \theta'$ hors de V. On a $\theta'' \circ \theta' = \theta'' \circ \theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1 = \theta_2'' \circ \theta_2 \circ \theta_1$. Soit $\alpha' \notin V$. D'après la remarque faite en début de preuve, les variables de V_1 sont hors de portée de θ_1 et donc $V_1 \cap \mathrm{VL}(\theta_1(\alpha')) = \emptyset$. Donc :

$$\begin{array}{rcl} (\theta'' \circ \theta')(\alpha') & = & (\theta''_2 \circ \theta_2 \circ \theta_1)(\alpha') \\ & = & (\theta''_1 \circ \theta_1)(\alpha') \\ & = & \theta(\alpha') \end{array}$$

D'où le résultat.

■

3.2 Simplification du langage

3.2.1 Annotations de types redondantes

Avec les restrictions que nous avons faites pour assurer la complétude de l'algorithme de type, une partie de l'information de type contenue dans l'écriture des expressions devient redondante : elle peut en effet être retrouvée à l'aide de l'algorithme de synthèse de type.

C'est en particulier le cas pour l'expression de création d'objets $\mathtt{new}^{\mathcal{P}}\ a$. Supposons en effet donné un environnement Γ. Comme le typage est principal, il existe une substitution θ_0 et un type τ_0 tels que $\theta_0(\Gamma) \vdash a : \tau_0$ et tels que, si $\theta(\Gamma) \vdash a : \tau$, alors il existe une substitution simple θ_1 telle que $\theta_1(\tau_0) = \tau$. Pour typer l'expression $\mathtt{new}^{\mathcal{P}}\ a$, on doit utiliser la règle de typage Classe-Création que nous rappelons ci-dessous :

$$\frac{\begin{array}{c}\Gamma \vdash a : Cl(\mathcal{L}, \tau', \mathcal{D}, \mathcal{P}) \\ \mathcal{D} = \mathrm{dom}\,\mathcal{L} \qquad \tau' = \langle \mathcal{L}|_{\mathcal{P}} \rangle\end{array}}{\Gamma \vdash \mathtt{new}^{\mathcal{P}}\ a : \tau'}$$

Il doit donc exister un type τ de la forme $Cl(\mathcal{L}, \tau', \mathcal{D}, \mathcal{P})$, et tel que $\tau = \theta_1(\tau_0)$. Comme θ_1 est simple, le type τ_0 est de la même forme, et possède de plus le même ensemble de méthodes publique $\mathcal{P}$. Cet ensemble est donc indépendant du jugement de typage $\theta(\Gamma) \vdash a : \tau$ considéré. L'annotation de l'expression $\mathtt{new}^{\mathcal{P}}\ a$ par un ensemble de méthodes publiques peut ainsi être retrouvée par l'algorithme de synthèse de type. Elle pourrait donc être omise.

Une simplification similaire est possible pour les coercions de classes $(a : \tau' < \tau)$. En effet, l'étude des règles de traduction définissant le jugement $\vdash \tau' < \tau \Rightarrow a$ (figure 2.11 de la section 2.3) montre que la fonction de traduction a est déterminée par la donnée de τ ainsi que des ensembles $\mathcal{D}$ des méthodes définies et $\mathcal{P}$ des méthodes publiques du type τ'. Comme nous venons de le voir, ces ensembles sont indépendants du type τ'' tel que $\theta(\Gamma) \vdash a : \tau''$. Il serait donc possible de n'exiger que la donnée du type $\tau : (a < \tau')$, les informations nécessaires à la compilation de cette expression pouvant être retrouvées par inférence de type.

3.2.2 Introduction des méthodes et méthodes publiques

Dans le corps des classes, les méthodes doivent être explicitement introduites et rendues publiques (constructions `abstract` l et `public` l). Nous montrons maintenant comment utiliser la synthèse de type afin de rendre ces constructions implicites.

L'introduction des méthodes est explicite afin que la traduction d'une classe en un prototype (donnée en section 2.3) ne dépende pas de son type. En effet, le domaine du dictionnaire d'un prototype doit être égal au domaine de la rangée $\mathcal{L}$ de son type $Prot(\mathcal{L}, \tau, \mathcal{D})$ (règle de typage Prototype donnée en section 2.2.4). Or cette rangée $\mathcal{L}$ est la même que celle du type de la classe $Cl(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P})$. Le domaine de cette rangée devrait donc être indépendante du typage. On peut cependant remarquer que le domaine du dictionnaire d'un prototype ou d'un objet n'est pas observable par la sémantique : le comportement d'un prototype ou d'un objet ne change pas si l'on étend son dictionnaire. Il suffit donc en fait que le dictionnaire du prototype obtenu contienne au moins toutes les méthodes utilisées dans le corps de la classe. On peut donc utiliser une rangée extensible ω comme en section 2.1.2 lors de la synthèse de type. Le domaine de la rangée ω du type synthétisé fournit alors un sur-ensemble des méthodes privées utilisées par la classe, ce qui permet de définir un dictionnaire φ suffisamment large pour le prototype.

Les méthodes doivent être rendues explicitement publiques pour une raison différente : le typage ne serait pas principal autrement. En effet, considérons la règle de typage concernant la création d'un objet :

$$\frac{\begin{array}{c}\Gamma \vdash a : Cl(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P}) \\ \mathcal{D} = \operatorname{dom} \mathcal{L} \qquad \tau = \langle \mathcal{L}|_{\mathcal{P}} \rangle\end{array}}{\Gamma \vdash \mathtt{new}^{\mathcal{P}}\ a : \tau} \ (\textsc{Classe-Création})$$

Si $\mathcal{P}$ n'était pas précisé, on pourrait dans certains cas choisir de rendre un plus ou moins grand nombre de méthodes publiques : le nombre minimal de méthodes publiques est donné par les méthodes de τ dans un typage principal $\theta(\Gamma) \vdash a : \tau$, mais un choix d'un type τ plus précis permet d'avoir plus de méthodes publiques. Cependant, une manière de garder le typage principal sans avoir à spécifier explicitement $\mathcal{P}$ est de dériver cet ensemble d'un typage principal $\theta(\Gamma) \vdash a : Cl(\mathcal{L}, \tau, \mathcal{D}, \mathcal{P})$ du corps a d'une classe : l'ensemble $\mathcal{P}$ peut être défini comme l'ensemble des méthodes apparaissant dans le type τ d'un tel jugement principal. Si le programmeur désire un plus grand nombre de méthodes publiques, il a toujours la possibilité d'ajouter une contrainte de type augmentant le nombre de méthodes de τ. On peut ainsi par exemple garder pour cela la construction `public` l, en ne la considérant que comme une annotation de type.

3.2.3 Simplification du type des classes

Le type d'une classe $Cl(\omega, \tau, \mathcal{D}, \mathcal{P})$ est relativement compliqué, et il serait intéressant de pouvoir n'en montrer qu'une version simplifiée à l'utilisateur. Nous supposons comme nous venons de le proposer que le type d'une classe contient une rangée ouverte ω et que dans un typage principal l'ensemble des méthodes publiques $\mathcal{P}$ du type d'une classe $Cl(\omega, \tau, \mathcal{D}, \mathcal{P})$ soit exactement les méthodes de τ.

Une première source de complication est que le type de la classe contient deux séquences de types de méthode ω et τ. On peut cependant remarquer que pour pouvoir créer un objet à partir d'une classe, les méthodes de τ doivent également être des méthodes de ω et doivent de plus avoir le même type dans τ et dans ω. Il serait intéressant d'avoir une contrainte similaire sur le type des classes parentes. C'est le cas si les règles définissant les contraintes de classes sont modifiées afin que les méthodes de ω ne puissent être masquées que si elles ne sont pas présentent dans τ. Alors dans toute classe, les méthodes communes à ω et τ doivent avoir des types compatibles. De plus, comme il faudra éventuellement que les méthodes de ω constituent un sur-ensemble de celles de τ (ce qui est toujours possible quitte à instancier le type ω), il paraît raisonnable d'imposer cette condition aux types présentés à l'utilisateur.

Avec ces conditions, les méthodes de τ sont exactement les méthodes de ω apparaissant dans $\mathcal{P}$. D'autre part, les règles de typage imposent que les méthodes définies $\mathcal{D}$ soient un sous-ensembe du domaine de ω. Il est alors possible d'afficher un type de classe simplement, on ne listant qu'une fois chaque méthode et en utilisant des annotations pour indiquer si elle est définie (c'est-à-dire si elle est dans $\mathcal{D}$), ou si elle est publique (c'est-à-dire si elle est dans $\mathcal{P}$).

3.3 Abréviations

Les abréviations sont habituellement juste une commodité pour le programmeur : elles lui évitent d'avoir à taper de nombreuses fois un type pouvant être large. Cependant, comme les types des objets peuvent être vraiment très larges, les abréviations s'avèrent critiques pour les afficher de manière lisible. Il est donc nécessaire d'étendre le mécanisme d'abréviations de ML pour qu'il fonctionne efficacement en présence de types d'objets et de types récursifs.

Il est ainsi important que les abréviations soient propagées autant que possible : si un type est abrégé, tous les types qui sont unifiés par la suite avec ce type doivent utiliser si possible cette même abréviation. D'autre part, une classe plus importante d'abréviations de types que celle usuellement disponible dans ML est nécessaire. Ainsi, il doit être possible d'abréger des types récursifs. De plus, il n'apparaît pas suffisant de simplement paramétrer les abréviations par des variables de type : on a besoin dans certains cas de limiter le type possible d'un paramètre aux instances d'un certain type. Ces deux points sont détaillés en section 3.3.1.

Le type d'un objet est en grande partie défini par le type de la classe à laquelle il appartient. Il est donc extrêmement pratique de s'appuyer sur le type de la classe afin de définir une abréviation pour le type de ses instances. Seules quelques annotations de type sont nécessaires, dans le cas des classes polymorphes. Le programmeur n'a ainsi pas à explicitement définir une abrèviation dans ce cas. De plus cette abréviation peut être utilisée pour le type du constructeur `new` a associé à la classe. Nous développerons ce point en section 3.3.2.

3.3.1 Abréviations avec contraintes

Les abréviations en ML peuvent être paramétrées par des variables de type. Nous avons parfois besoin que ce paramètre soit contraint à être une instance d'un certain type. Supposons en effet définies les abréviations suivantes, qui représentent le type d'un point et d'un point coloré :

```
type point = < position : int * int >
type point_coloré = < position : int * int; couleur : string >
```

Essayons donc de définir une abréviation pour un cercle dont le centre est représenté par un objet ayant au moins les méthodes d'un point. Le type d'un cercle a donc une variable libre : la variable de rangée du type du centre. On pourrait paramétrer l'abréviation décrivant le type d'un cercle par cette variable de rangée :

```
type 'a cercle =
          < centre : < position : int * int; 'a >; rayon : int >
```

Mais alors, un cercle dont le centre est un point coloré aurait le type :

```
             (couleur:string) cercle
```

Ce type n'est pas très intuitif. En effet, une portion de rangée isolée n'a pas tellement de sens et on aimerait plutôt pouvoir parler du type du point dans son intégralité. De plus, l'abréviation `point_coloré` ne peut pas être utilisée pour rendre ce type plus concis[2]. On aimerait donc que le paramètre de l'abréviation soit le type du point en entier :

```
type 'a cercle = < centre : 'a; rayon : int >
```

Le type d'un cercle dont le centre est un point coloré est alors bien

```
             point_coloré cercle
```

Mais cette abréviation est plus générale que la précédente : son paramètre pourrait être le type `int`, par exemple. On aimerait pouvoir spécifier lorsque l'on définit l'abréviation cercle que son parametre n'est pas arbitraire, mais a bien une certaine forme (c'est-à-dire, est une instance d'un certain type), ce que l'on pourrait écrire :

```
type 'a cercle = < centre : 'a; rayon : int >
         constraint 'a = < position : int * int; 'b >
```

De manière plus formelle, une abréviation t serait définie par un ensemble de type $\vec{\tau}$ constituant les paramètres de l'abréviation (et les contraintes qui leur sont associées) ainsi que par un type τ_0 spécifiant son expansion : $\forall\vec{\alpha}.((\vec{\tau})\ t = \tau_0)$ (nous avons quantifié l'équation afin qu'elle soit close, c'est-à-dire qu'il n'y ait pas de variable libre). La sémantique voulue est que l'expansion de $(\theta(\vec{\tau}))\ t$ est $\theta(\tau_0)$. Il est bien entendu nécessaire d'imposer que les variables de τ_0 soient des variables apparaissant dans $\vec{\tau}$, afin que $\theta(\tau_0)$ puisse être déduit des types $\theta(\vec{\tau})$. Cette sémantique est une généralisation de la sémantique usuelle des abréviations.

Montrons maintenant comment l'unification sur des types avec abréviations peut être définie. Les constructeurs de types sont ajoutés à la grammaire des types : $(\vec{\tau})\ t$. On suppose que l'expansion de chaque constructeur est spécifiée par une équation $\forall\vec{\alpha}.(\vec{\tau})\ t = \tau_0$. L'algorithme est une modification de l'algorithme d'unification sur des graphes présenté en section 1.4.4. Nous reprenons cet algorithme en ajoutant une seconde forme d'équation $\alpha \equiv ({\alpha_i}^{i\in I})\ t$ signifiant que le type α est l'expansion de l'abréviation t. Nous ajoutons également une règle Expansion définissant l'expansion d'une abréviation : un constructeur de type est remplaçé par son expansion ; de plus, on garde une trace du lien entre le constructeur et son expansion en utilisant la seconde forme d'égalité.

[2] Il y a en fait un troisième problème. En effet, pour pouvoir définir de telles abréviations, il devient nécessaire de donner une sorte aux paramètre des constructeurs de types. La sorte peut être inférée dans le cas des abréviations, mais devrait être donner explicitement lors de la définition d'un type abstrait.

(Fusionne)
$$\frac{\alpha \doteq e \wedge \alpha \doteq e'}{\alpha \doteq e \doteq e'}$$

(Décompose)
$$\frac{f({\alpha_i}^{i\in I}) \doteq f({\alpha'_i}^{i\in I}) \doteq e}{f({\alpha_i}^{i\in I}) \doteq e \wedge (\alpha_i \doteq \alpha'_i)^{i\in I}}$$

(Généralise)
$$\frac{e[\alpha \leftarrow \tau] \qquad \alpha \notin \mathrm{VL}(\tau)}{\exists\, \alpha.\, e \wedge \alpha \doteq \tau}$$

(Commute)
$$\frac{(m_1 : \alpha_1; \alpha'_1) \doteq (m_2 : \alpha_2; \alpha'_2) \doteq e}{\exists\, \alpha'.\, (m_2 : \alpha_2; \alpha'_2) \doteq e \wedge \alpha'_1 \doteq (m_2 : \alpha_2; \alpha') \wedge \alpha'_2 \doteq (m_1 : \alpha_1; \alpha')}$$

(Expansion)
$$\frac{({\alpha_i}^{i\in I})\; t \doteq e \qquad \forall \vec{\alpha}.({\tau_i}^{i\in I})\; t = \tau_0}{\exists\, \alpha_0.\, \exists\, \vec{\alpha}.\, \alpha_0 \equiv ({\alpha_i}^{i\in I})\; t \wedge \alpha_0 \doteq \tau_0 \doteq e \wedge {\alpha_i}^{i\in I} \doteq {\tau_i}^{i\in I}}$$

(1) Dans la règle Décompose, f est n'importe quel symbole de type, y compris $\_ \to \_$ et $(m : \_; \_)$.
(2) Afin d'assurer la terminaison, la règle Généralise doit être restreinte au cas où τ n'est pas une variable de type et α apparaît dans la multi-équation e mais aucun des termes de e n'est α.

Fig. 3.5: Règles d'unification

Nous supposons que l'expansion et les paramètres d'une abréviation t ne contient pas d'occurrence du constructeur t, et que, de plus, les abréviations que ces types contiennent ne font pas non plus intervenir le type t. Ces conditions assurent la terminaison de l'unification. Elles ne sont pas très restrictives. Il est en effet possible de réécrire une définition (1) $t = \langle m : t \rangle$ en une définition bien formée équivalente (2) $t = \mu(\alpha)\langle m : \alpha \rangle$, dans le sens où le type t vérifiant (1) est le type donné par (2). Dans le cas d'une abréviation possédant des paramètres de type, la restriction se traduit pas une contrainte de linéarité : les occurences de l'abréviation dans son expansion doivent posséder les mêmes paramètres de type que l'abréviation elle-même.

Montrons par exemple comment se déroule l'unification des types $t * \alpha'$ et $\langle m : \alpha \rangle * \alpha$, où le type t s'expanse en $\langle m : int \rangle$. Nous omettrons la quantification des variables que nous sommes amener à introduire. On part donc du système suivant (la variable de type α_0 désigne le type obtenu après unification) :

$$\alpha_0 \doteq (t * \alpha') \doteq (\langle m : \alpha \rangle * \alpha)$$

On commence par appliquer la règle Généralise.

$$\alpha_0 \doteq (\alpha_1 * \alpha') \doteq (\alpha_2 * \alpha) \wedge \alpha_1 \doteq t \wedge \alpha_2 \doteq \langle m : \alpha \rangle$$

On peut alors appliquer la règle Décompose.

$$\alpha_0 \doteq (\alpha_1 * \alpha') \wedge \alpha_1 \doteq \alpha_2 \doteq t \doteq \langle m : \alpha \rangle \wedge \alpha' \doteq \alpha$$

La règle Expansion donne alors.

$$\alpha_0 \doteq (\alpha_1 * \alpha') \wedge \alpha_1 \doteq \alpha_2 \doteq \alpha_3 \doteq \langle m : int \rangle \doteq \langle m : \alpha \rangle \wedge \alpha' \doteq \alpha \wedge \alpha_3 \equiv t$$

On applique alors à plusieurs reprises les règles GÉNÉRALISE et DÉCOMPOSE. D'abord la règle GÉNÉRALISE :

$$\begin{aligned}&\alpha_0 \doteq (\alpha_1 * \alpha') \wedge \alpha_1 \doteq \alpha_2 \doteq \alpha_3 \doteq \langle \alpha_4 \rangle \doteq \langle \alpha_5 \rangle \wedge \\ &\alpha_4 \doteq (m : int) \wedge \alpha_5 \doteq (m : \alpha) \wedge \alpha' \doteq \alpha \wedge \alpha_3 \equiv t\end{aligned}$$

Règle DÉCOMPOSE :

$$\begin{aligned}&\alpha_0 \doteq (\alpha_1 * \alpha') \wedge \alpha_1 \doteq \alpha_2 \doteq \alpha_3 \doteq \langle \alpha_4 \rangle \wedge \\ &\alpha_4 \doteq \alpha_5 \doteq (m : int) \doteq (m : \alpha) \wedge \alpha' \doteq \alpha \wedge \alpha_3 \equiv t\end{aligned}$$

Règle GÉNÉRALISE :

$$\begin{aligned}&\alpha_0 \doteq (\alpha_1 * \alpha') \wedge \alpha_1 \doteq \alpha_2 \doteq \alpha_3 \doteq \langle \alpha_4 \rangle \wedge \alpha_4 \doteq \alpha_5 \doteq (m : \alpha_6; \alpha_7) \doteq (m : \alpha; \alpha_8) \wedge \\ &\alpha_6 \doteq int \wedge \alpha_7 \doteq \emptyset \wedge \alpha_8 \doteq \emptyset \wedge \alpha' \doteq \alpha \wedge \alpha_3 \equiv t\end{aligned}$$

Règle DÉCOMPOSE :

$$\begin{aligned}&\alpha_0 \doteq (\alpha_1 * \alpha') \wedge \alpha_1 \doteq \alpha_2 \doteq \alpha_3 \doteq \langle \alpha_4 \rangle \wedge \alpha_4 \doteq \alpha_5 \doteq (m : \alpha_6; \alpha_7) \wedge \\ &\alpha' \doteq \alpha = \alpha_6 \doteq int \wedge \alpha_7 = \alpha_8 \doteq \emptyset \wedge \alpha_3 \equiv t\end{aligned}$$

On obtient ainsi un système sous forme résolu. Ce système d'équations contient de nombreuses équations inutiles. Pour des raisons de performances, et afin de présenter un type plus simple à l'utilisateur, on veut garder le moins possible d'équations. Tout d'abord, il est clair que les variables de type n'apparaissant qu'une fois peuvent être éliminées :

$$\alpha_0 \doteq (\alpha_1 * \alpha') \wedge \alpha_1 \doteq \langle \alpha_4 \rangle \wedge \alpha_4 \doteq (m : \alpha; \alpha_7) \wedge \alpha' \doteq \alpha \doteq int \wedge \alpha_7 \doteq \emptyset \wedge \alpha_1 \equiv t$$

On peut alors remarquer que l'équation $\alpha_1 \doteq \langle \alpha_4 \rangle$ pourrait être dérivée de l'équation $\alpha_1 \doteq t$, par expansion de l'abréviation. Plus généralement, pour chaque équation $\alpha \equiv ({\alpha_i}^{i \in I})\ t$, on voudrait éliminer les égalités dont l'un des membres est α et s'appuyer sur l'expansion de l'abréviation pour retrouver ces égalités. Cela donne ici (la variable α_4 qui n'apparaîtrait alors qu'une fois a également été éliminée) :

$$\alpha_0 \doteq (\alpha_1 * \alpha') \wedge \alpha' \doteq \alpha \doteq int \wedge \alpha_1 \doteq t$$

On peut alors lire aisément la solution :

$$\begin{cases} \alpha_0 = t * int \\ \alpha = \alpha' = int \end{cases}$$

Éliminer des égalités comme nous venons de le faire n'est cependant pas toujours correct. Considérons donc par exemple l'abréviation définie par $\alpha\ t = \alpha$ et essayons d'unifier en particulier le type $int\ t$ avec son argument : $\alpha \doteq int \doteq \alpha\ t$. On obtient l'équation sous forme résolue suivante : $\alpha \doteq int \wedge \alpha \equiv \alpha\ t$. La première égalité ne peut pas être supprimée ici, car le fait que l'expansion de l'abréviation est int serait perdu.

On peut remarquer qu'ici, si l'on ne considère que l'équation $\alpha \doteq \alpha\ t$ et que l'on lui applique les règles d'unification, on obtient un système $\alpha \equiv \alpha\ t$ dans lequel la variable α n'est pas instanciée (il n'y a pas d'équation $\doteq$ dont l'un des membres est cette variable), contrairement au système sous forme résolu précédent. Cela nous donne en fait un critère complet permettant de déterminer si une équation peut être éliminée : elle peut l'être si et seulement si le nombre de variables non instanciées n'augmente pas.

Une tel critère est cependant très coûteux à utiliser, car il nécessite une nouvelle expansion des abréviations après unification. On peut cependant utiliser un critére plus rapide mais non complet : on peut supprimer une égalité $\alpha \doteq \tau$ si l'on a une équation $\alpha \equiv ({\alpha_i}^{i \in I})\ t$ et α n'est accessible par aucun des α_i, c'est-à-dire si les types représentés par les α_i ne contiennent pas α. Dans l'exemple précédent, il est alors évident que la simplification n'est pas possible car α est l'un des paramètres. Dans les rares cas où cette heuristique échoue, il est toujours possible d'appliquer le critère complet.

3.3.2 Définition implicite d'abréviations

Il serait pénible d'avoir à définir explicitement une abréviation pour le type des objets de chaque classe. Cependant, comme le type des classes, qui est inféré, spécifie assez précisément le type de leurs instances, il est possible d'en dériver automatiquement des abréviations.

Le type du corps d'une classe est de la forme $(\mathcal{L}, \tau_0, \mathcal{D}, \mathcal{P})$. D'après la règle de traduction de **new**, le type d'un objet de la classe est une instance de τ_0 dont les méthodes sont les méthodes publiques $\mathcal{P}$ de la classe.

On peut imposer que τ_0 soit de la forme $\langle l : {\tau_l}^{l \in \mathcal{P}}; \tau' \rangle$. où τ' est soit une variable de rangée soit le type vide. Toute instance de la classe ou de l'une de ses sous-classes ayant au moins comme méthodes publiques les méthodes de $\mathcal{P}$, cette contrainte n'est guère restrictive.

Supposons que τ' soit une variable de rangée ρ. Supposons de plus que τ_0 ne contienne pas de variables autres que ρ. On peut alors définir une abréviation $\#t$ pour le type de la classe :

$$\rho\ \#t = \tau_0$$

Cette abréviation donne la forme du type d'un objet de cette classe ou de n'importe laquelle de ses sous-classes. On peut également définir une abréviation t spécifiquement pour le type des instances directes de cette classe :

$$t = (\emptyset)\#t$$

Cette dernière abréviation peut par exemple être utilisée pour abréger le type du constructeur associé à la classe. Dans le cas où τ' est le type vide, on peut donner la même définition aux deux abréviations[3] :

$$t = \#t = \tau_0$$

Le type τ_0 peut avoir d'autres variables libres que ρ. Les abréviations doivent alors être paramétrées, et on a donc besoin d'un moyen de spécifier quels sont ces paramètres. Considérons donc la classe suivante :

```
class cercle x = object
  method centre = x
```

[3]En fait, dans Objective Caml, les classes dont le type τ_0 n'est pas extensible ne sont pas autorisées : en effet, l'expérience a montré que ce genre de classe n'était pas utile, et que leur comportement (une sous-classe ne peut pas ajouter de nouvelles méthodes) était difficile à comprendre pour l'utilisateur.

```
  method position = x#position
end
```

On a $\tau_0 = \langle \texttt{centre} : \langle \texttt{position} : \alpha; \rho' \rangle; \texttt{position} : \alpha; \rho \rangle$, où $\langle \texttt{position} : \alpha; \rho' \rangle$ est le type du paramètre `x` de la classe. Une abréviation de type $\rho\ \#t = \tau_0$ ne serait pas correcte car les variables α et ρ' ne seraient pas liées par le paramètres de l'abréviation. La classe doit donc être annotée pour que l'on puisse en dériver des abréviations :

```
class ['a] cercle (x : 'a) = object
  method centre = x
  method position = x#position
end
```

La classe comprend maintenant une liste de paramètres de type, entre crochets. Des annotations de types permettent d'associer à chacun de ces paramètres une partie du type de la classe (ici, le type $\langle \texttt{position} : \alpha; \rho' \rangle$ de la variable x est associé au paramètre `'a`). Il est alors possible de définir les abréviations de la classe :

$$\begin{array}{rcl} (\tau_1, \rho)\ \#\texttt{cercle} & = & \langle \texttt{centre} : \tau_1; \texttt{position} : \alpha; \rho \rangle \\ \tau_1\ \texttt{cercle} & = & \langle \texttt{centre} : \tau_1; \texttt{position} : \alpha \rangle \end{array}$$

où $\tau_1 = \langle \texttt{position} : \alpha; \rho' \rangle$. Le type de l'expression `new cercle` est alors $\tau_1 \rightarrow \tau_1\ \texttt{cercle}$.

Plus précisément, une définition de classe comprend une liste de paramètres de type α_i. Ces paramètres de type α_i sont instanciés lors de la synthèse de type à des types τ_i. Les abréviations associées à la classe sont alors définies par

$$\begin{array}{rcl} (\tau_1, \ldots, \tau_n, \rho)\ \#t & = & \tau_0 \\ (\tau'_1, \ldots, \tau'_n)\ t & = & (\tau'_1, \ldots, \tau'_n, \emptyset)\ \#t \end{array}$$

En pratique, on a souvent besoin d'utiliser les abréviations associées à une classe à l'intérieur de la définition de cette classe, notamment pour abréger le type du constructeur `new` s'il est utilisé à l'intérieur de la classe. Nous allons maintenant voir comment permettre cela même si les abréviations ne sont alors pas encore complètement définies. Afin de simplifier la présentation, nous supposons que la classe ne comprend pas de paramètre de type (l'extension au cas général est immédiate). On peut directement définir l'abréviation $\#t$ comme l'abréviation dont l'expansion est τ_0. L'abréviation t du type des instances de la classe est initialement une variable de type fraîche. Au cours de la synthèse de type, cette variable peut être instancié. On obtient ainsi un type τ_t. On veut avoir l'égalité suivante entre les abréviations associées à la classe : $t = (\emptyset)\#t$. On doit donc trouver une substitution θ telle que

$$\theta(\tau_t) = \theta(\tau_0)\{\emptyset/\rho\}$$

où ρ est la variable de rangée du type $\theta(\tau_0)$. Comme il s'avère difficile de résoudre cette équation directement, nous allons faire deux hypothèses supplémentaires.

La première est de supposer que $\theta(\rho) = \rho$. Cette contrainte n'est pas restrictive en pratique. En effet, si l'équation ci-dessus admet une solution θ_1, on vérifie aisément que cette même équation dans laquelle τ_0 est remplacé par $\tau_0\{\theta_1(\rho_0)/\rho_0\}$ (où ρ_0 est la variable de rangée de τ_0) admet une solution θ_2

telle que $\theta_2(\rho) = \rho$ (le point clé est que τ_t ne contient pas la variable ρ_0 car le type $\theta(\tau_t)$ ne contient pas la variable ρ). L'utilisateur peut donc ajouter des annotations à la définition de la classe afin que l'équation ait une solution (en pratique, ces annotations sont naturellement ajoutées par l'utilisateur, qui ne s'aperçoit donc pas de la difficulté).

La seconde hypothèse supplémentaire est que, pour toute variable ρ', si $\theta(\rho') = \rho$ alors $\rho' = \rho$. On peut justifier cette seconde hypothèse en remarquant que si l'on a une solution telle que $\theta(\rho') = \rho$ et $\rho' \neq \rho$, alors la substitution $\theta' = \theta; (\rho' = \emptyset)$ est également solution. En effet, le type $\theta(\tau_t)$ ne contient pas la variable ρ, et donc, d'après la première hypothèse, τ_t ne contient pas non plus cette variable. Il paraît donc raisonnable de ne pas prendre en compte les solutions qui unifient arbitrairement des variables de type à ρ.

Avec ces deux hypothèses supplémentaires, les solutions de $\theta(\tau_t) = \theta(\tau_0)\{\emptyset/\rho\}$ sont les mêmes que celles de $\theta(\tau_t) = \theta(\tau_0\{\emptyset/\rho\})$. De plus si la seconde équation admet une solution, on vérifie aisément qu'elle admet une solution équivalente satisfaisant les hypothèses supplémentaires. Enfin, cette dernière équation est facile à résoudre : la variable de rangée ρ est généralisée dans le type τ_0, puis instanciée au type vide. On obtient ainsi le type $\tau_0\{\emptyset/\rho\}$. Il n'y alors plus qu'à unifier ce type avec le type τ_t.

Chapitre 4

Réalisation pratique

Nous décrivons dans ce chapitre la compilation des constructions liées aux objets dans Objective Caml. Dans un premier temps, nous présentons la compilation des appels de méthode. Puis nous décrivons la compilation des classes.

4.1 Appels de méthodes

Il paraît à première vue difficile de compiler efficacement les appels de méthodes dans Objective Caml. En effet, on dispose de très peu d'information au moment de la compilation : ainsi, pour une fonction `let f x = x#m`, la seule information que l'on ait est qu'à l'exécution l'objet représenté par la variable `x` aura une méthode `m`. Mais l'objet peut appartenir à n'importe quelle classe. Au contraire, dans des langages comme C++ ou Java, lors d'un appel de méthode, la classe à laquelle appartient l'objet est connue[1]. En effet, le programmeur doit fournir explicitement le type statique de l'objet, et ce type est une classe. On peut donc traduire le nom de méthode en index dans la table des méthodes de cette classe. Nous allons voir que l'on peut utiliser un schéma presque aussi efficace pour Objective Caml.

On cherche donc à associer à un objet et un nom de méthode une fonction. Les méthodes des objets d'une même classe sont partagées. Les objets d'une même classe contiennent donc un pointeur vers une structure de donnée commune, donnant accès aux méthodes de la classe. On associe globalement, dans l'ensmeble du programme, à chaque nom de méthode un entier servant d'index dans cette table. Cette association est effectuée à l'exécution : une table associant aux noms de méthodes leurs index est tenue à jour. Lors de l'initialisation d'un module, la table est consultée afin d'obtenir les index des méthodes utilisées par ce module. Les index pourraient également être calculés lors de la compilation, au prix d'une plus grand complexité de l'éditeur de lien.

Comme les index des méthodes d'un objet ne sont souvent pas consécutifs, les utiliser directement pour accéder via un tableau aux méthodes consommerait beaucoup trop de mémoire. Mais une table creuse peut être réalisée en utilisant deux niveaux de tableaux (voir figure 4.1). Les index sont divisés en deux : leur partie haute est utilisée pour le tableau principale, et leur partie basse pour un

[1] En fait, un problème similaire se pose pour les interfaces en Java ; l'appel de méthode dans ce cas est alors plus lent.

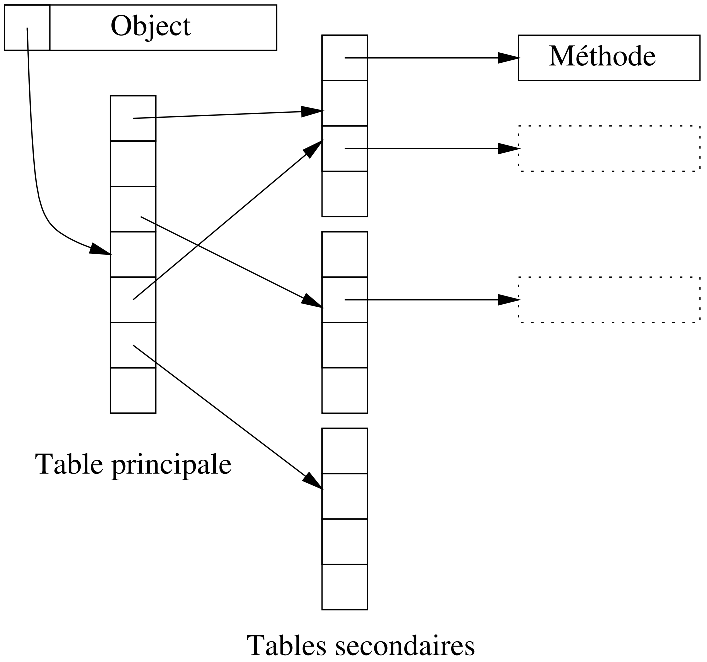

FIG. 4.1: Accés aux méthodes d'un objet

tableau secondaire. Ainsi, un appel de méthode `obj#meth arg1 .. argn` est compilé en une expression de la forme :

```
obj.(0).(idx / N).(idx mod N) obj arg1 ... argn
```

où `idx` est l'index associé à la méthode `m`. La taille des tableaux reste donc raisonnable. De plus, plusieurs tableaux secondaires peuvent être partagés si les méthodes qu'ils contiennent sont à des positions distinctes. Cette méthode de compilation des appels de méthodes est due à Kresten Thorup et est utilisée dans le compilateur Gnu Objective C.

La détermination de la méthode à appeler n'est ainsi que 50% plus lente qu'en Java ou C++. Elle se fait en temps constant, et par rapport à ces langages, elle ne nécessite qu'une indirection supplémentaire (deux indirections au lieu de trois). Le fait que les index ne soient pas fixés statiquement n'est pas un problème avec les processeurs modernes. En effet, le coût d'un appel de méthode y est esssentiellement déterminé par celui des indirections successives à effectuer, l'accès à la valeur de l'index et les différents calculs effectués sur celle-ci pouvant se faire en parallèle. Il ne paraît donc pas utile de compliquer l'édition de lien en choissant les index à ce moment.

Cependant, le coût d'un appel de méthode reste beaucoup plus important que celui de l'appel à une fonction connue, c'est-à-dire une fonction dont le compilateur peut déterminer statiquement l'adresse du code. En effet, lorsque la fonction n'est pas connue des indirections supplémentaires sont nécessaires afin d'accéder dans la fermeture de la fonction à son adresse. De plus, une fonction auxiliaire est d'abord appelée, qui détermine si le nombre d'arguments passés correspond au nombre attendu et crée une nouvelle fermeture le cas échéant. Pour réduire ce coût, il faudrait faire une analyse globale assez fine du programme, qui permettrait de remplacer un appel de méthode dynamique par l'appel direct de la fonction lorsqu'une seule fonction peut être appelée.

Le mécanisme que nous avons décrit s'applique aux méthodes publiques. Mais il peut être étendu aisément aux méthodes privées : il suffit d'associer à celles-ci des index spécifiques ne correspondant pas à des noms de méthodes.

4.2 Compilation des classes

L'initialisation des classes a lieu au début de l'exécution des programmes plutôt que lors de la compilation ou l'édition de lien. Ce schéma de compilation permet de minimiser les modifications apportées au compilateur et l'éditeur de lien. Il permet également une compilation séparée des différents modules formant un programme. Le résultat de la compilation d'une classe doit donc fournir suffisamment d'informations non seulement pour créer des instances de cette classe mais aussi pour initialiser ses sous-classes.

La formalisation que nous avons présenter au chapitre 2 page 27 suggère de représenter une classe comme un prototype (section 2.2 page 2.2) : le dictionnaire φ donnerait la correspondance entre le nom des méthodes et leur index dans la table L des méthodes de la classe. Cependant cette représentation nécessite lors de l'héritage multiple un renommage du dictionnaire de l'un des prototypes. Pour des raisons d'efficacité, un tel renommage doit être évité. Intuitivement, il faudrait pour cela que le dictionnaire du premier prototype soit connu avant la création du dictionnaire du second. Ainsi, on peut choisir pour le second dictionnaire des index qui n'interfèrent pas avec ceux du premier. Une classe est donc compilée comme une fonction. Cette fonction prend en argument une description partielle d'une sous-classe en cours d'initialisation. Elle complète la description en y ajoutant les méthodes et les variables d'instance de la classe. Les informations nécessaires à la création d'une instance sont alors obtenues en appliquant la fonction à une description vide. Cette application n'est bien sûre faite qu'une seule fois, lors de l'initialisation de la classe. Considérons donc par exemple la classe suivante :

```
class d = object
  inherit c 3
  val x = 1
  method m = x
end
```

Cette classe se traduit approximativement en la fonction suivante :

```
let class_init desc =
  let init_parent = c.class_init desc in
  let x_pos = Oo.new_variable desc "x" in
  Oo.set_method desc "m" (fun self -> self.(x_pos));
  fun self -> init_parent self 3; self.(x_pos) <- 1; self
```

La fonction commence par appeler la fonction d'initialisation de la classe parente afin d'ajouter à la description `desc` les méthodes et variables d'instance de cette classe. Une fonction `init_parent` permettant d'initialiser les variables d'instance de la classe parente est alors retournée. Puis une variable d'instance `x` est ajoutée à la description de la classe par l'appel de la fonction `Oo.new_variable`.

Cette fonction retourne la position de la variable d'instance dans l'objet. Une méthode `m` est ensuite définie. Le corps de cette méthode fait référence à la variable d'instance `x` et utilise donc la variable `x_pos` pour obtenir sa position dans l'objet `self`. Finalement, une fonction permettant l'initialisation des objets de la classe est retournée. Celle-ci appelle `init_parent` afin d'initialiser les variables d'instance de la classe parente, puis fixe la valeur de la variable `x`.

Détaillons plus précisement certains points. Comme les méthodes publiques ne peuvent pas être masquées, l'ensemble des méthodes publiques d'une classe et de ses super-classes est connu lors de l'initialisation de cette classe. La description de la classe `desc` peut donc être initialisé avec cet ensemble. La fonction `Oo.set_method` peut ainsi déterminer si la méthode qu'elle ajoute est publique ou privée.

Une contrainte de classe $(a : \tau < \tau')$ permet de masquer certaines méthodes et variables d'instance ainsi que d'oublier la définition d'autres méthodes. Une méthode masquée ainsi ne doit pas avoir le même index dans la table des méthodes qu'une méthode de même nom apparaissant dans une sous-classe. De manière similaire, si la définition d'une méthode est oubliée, elle ne doit pas remplacer une définition donnée précédemment dans une sous-classe. Pour cela, la description `desc` d'une classe comprend l'index des méthodes déjà définies et l'ensemble des méthodes dont la définition doit être ignorée. Une classe soumise à une contrainte est alors compilée en une fonction qui modifie ces valeurs, appel la fonction d'initialisation de la classe et restaure les valeurs précédentes :

```
fun desc ->
  Oo.narrow desc ...;
  let init = ... in
  Oo.widen desc;
  init
```

Il est possible d'optimiser l'initialisation d'une classe lorsque le corps de celle-ci commence par de l'héritage : plutôt que d'appeler la fonction d'initialisation du parent, on peut en effet directement réutiliser les informations obtenues précédemment par l'application de cette fonction à une description vide. Il faut cependant prendre en compte le fait qu'une méthode peut être privée dans la classe parente et publique dans la sous-classe : deux index devront donc être associés à une telle méthode. En effet, les méthodes de la classe parente sont compilées pour appeler cette méthode en utilisant un index privé, et elle doit également pouvoir être appelé par l'index publique correspondant à son nom.

Décrivons maintenant la compilation de la construction `new c`. Après initialisation de la classe `c`, on connaît les informations nécessaires à la création des objets de cette classe, à savoir la taille de ses objets, la table des méthodes et une fonction initialisant les variables d'instance. On pourrait penser qu'il suffit pour créer un objet de l'allouer, faire pointer son premier champ vers la table des méthodes de sa classe, puis l'appliquer à la fonction d'initialisation. Cependant, cette dernière application est généralement partielle, et la fonction retournée peut très bien être appliquée plus tard successivement à des arguments différents. Elle retournera alors à chaque fois le même objet, initialisé de nouveau, alors qu'elle devrait retourner un nouvel objet. L'objet doit donc être alloué dans la fonction d'initialisation elle-même. La compilation de la fonction d'initialisation présentée plus haut doit donc être modifiée ainsi :

```
fun self arg1 ... argn ->
```

```
let self = if self <> () then self else <allouer self> in
...
```

La construction `new c` applique alors simplement `()` à cette fonction d'initialisation.

Chapitre 5

Un calcul d'objets avec des vues

Afin de rendre les programmes plus lisibles et plus faciles à maintenir, il est important de pouvoir subdiviser ceux-ci, et que chaque partie ne donne directement accès qu'à un sous-ensemble de son contenu aux autres parties. Les langages à objets fournissent habituellement des mécanismes pour cela. Par exemple, la définition d'une méthode ou d'une variable d'instance peut être annotée de manière à en limiter la portée : les composants privés d'une classe ne peuvent être accédés que depuis des méthodes définies dans la même classe. Cependant, dans ce cas, le choix de rendre un composant privé est fait *a priori*. Par conséquent, l'ajout d'une méthode publique à une classe peut nécessiter la modification de ses sous-classes, et du code dépendant de ces sous-classes, afin de l'adapter à ce changement. En effet, une sous-classe peut déjà contenir une méthode de même nom. Cela est bien peu satisfaisant, et on aimerait pouvoir étendre l'interface d'une classe sans interférer avec la définition de ses sous-classes. En d'autres termes, il devrait être possible de masquer les composants d'une classe *a posteriori* de manière à ce que chaque sous-classe puisse sélectionner une portion fixe de l'interface de sa classe parente. Riecke et Stone [30] ont étudié ce problème et en ont donné une solution élégante. Leur travail ne prend cependant pas en compte les méthodes binaires, et leur approche ne semble pas pouvoir s'étendre à ce cas.

En s'appuyant sur leurs idées, nous avons étudié une autre approche, qui prend en compte les méthodes binaires dès le début. Le calcul de Riecke et Stone repose essentiellement sur deux mécanismes. Pour la sémantique, un accès privilégié depuis le corps d'une méthode aux autres méthodes de l'objet courant permet de rajouter une méthode sans perturber les liaisons entre méthodes déjà existantes. Pour le typage, le masquage des méthodes est obtenu par sous-typage. Comme les méthodes binaires doivent pouvoir accéder aux méthodes d'un autre objet que l'objet courant, ces mécanismes ne peuvent être utilisés. En effet, un accès privilégié à un objet n'est possible que si la structure de cet objet est connue en détail. Mais on veut justement pouvoir donner aux objets des types qui ne décrivent qu'une partie de leur interface. Voici par exemple le type d'un objet possédant une méthode binaire : `Obj('a)[x: int; compare: 'a -> bool]`. C'est le type d'un objet ayant une méthode `x` retournant un entier et une méthode `compare` prenant un argument de même type que cet objet et retournant un booléen. La méthode `compare` peut tout à fait appeler la méthode `x` de son argument. S'il était possible de masquer la méthode `x` par sous-typage dans une sous-classe, le type d'un objet de cette sous-classe serait `Obj('a)[compare: 'a -> bool]`. Mais alors, rien n'empêcherait de passer un objet de même type à la méthode `compare`, ce qui est clairement incorrect : rien ne garantit qu'un tel objet ait une méthode `x`. Heureusement, d'autres mécanismes peuvent être utilisés. Le renommage permet d'éviter les conflits de noms et les types existentiels permettent de masquer de l'information. Les noms de méthodes sont globaux et ne peuvent pas être soumis à l'α-conversion. Nous introduisons donc une nouvelle notion : les *vues*. À chaque objet, nous associons un ensemble de vues. En plus d'accéder directement à une méthode d'un objet par son nom, il est possible d'y accéder en utilisant une vue. Les vues sont explicitement introduites et ont donc une portée limitée et peuvent être renommées librement. Le masquage des vues se fait en représentant un ensemble de vues par une seule vue à l'aide de types existentiels.

Nous présentons un calcul basé sur ces idées. Ce calcul n'a pas de notion de classe : les objets sont dérivés les uns des autres par ajout, masquage et redéfinition de méthodes. Afin de fournir une présentation plus claire et montrer que ce calcul peut être utilisé comme la base d'un vrai langage, nous avons également défini une extension du calcul avec des classes.

La notion de vue est intéressante en elle-même. En effet, comme les calculs d'objets utilisent généralement le sous-typage pour masquer de l'information, les composants privés d'un objet ne sont accessibles que depuis les méthodes de cet objet. Au contraire, dans les langages à objets courants tels que Java ou C++, les champs privés d'un objet sont aussi souvent accessibles depuis d'autres objets de la même classe, ou depuis des fonctions *amies*. Les vues permettent de modéliser cette fonctionnalité.

5.1 Exemples

5.1.1 Classes simples

Nous commençons par donner quelques exemples écrits dans le langage complet. Une classe est similaire à une fonction prenant en argument une valeur, ici `x0`, et retournant un objet. Cet objet est décrit par le corps de la classe. La classe suivante définit des points. Elle a une seule méthode `position`, qui retourne la valeur du paramètre de classe `x0`.

```
class point_simple (x0 : int) = object
  method position : int = x0
end
```

Les objets de cette classe sont créés par l'expression `new point_simple`, de type `int -> Obj('a)[position: int]` : c'est une fonction prenant un entier en argument et retournant un objet. Le type de cet objet est paramétré par une variable de type `'a` représentant le type lui-même. Il liste les méthodes de l'objet.

Une vue est un nom donnant accès à une interface alternative d'un objet. Les méthodes peuvent être appelées ou redéfinies soit directement, soit via une vue (en utilisant la syntaxe `o.[v]m`, où `o` est un objet, `v` est une vue et `m` est un nom de méthode). Depuis une méthode d'un objet, les autres méthodes de cet objet ne peuvent être appelées que par une vue. En effet, dans notre calcul, l'interface principale des instances de la classe n'est pas connue, car elle peut être complètement modifiée par héritage, en cachant certaines méthodes et en en rajoutant d'autres. Par contre, l'ensemble des vues ne peut qu'être étendu par héritage. Les vues sont introduites par les définitions de classes : dans l'exemple ci-dessous, une vue `v` est introduite au début de la classe. Cette vue est locale à la classe et permet aux méthodes de la classe de s'appeler mutuellement. La variable `moi` introduite au début de la classe avec son type `'a` représente un objet de la classe. La méthode `position` est la même que celle de la classe précédente. La méthode `déplace` retourne une copie de l'objet `moi` où la méthode `position` a été modifiée de manière à retourner la valeur de `x'` plutôt que celle du paramètre de la classe. Ces deux méthodes modélisent une variable d'instance mutable.

```
class point [v] (x0 : int) = object (moi : 'a)
  method position : int = x0
  method déplace (x' : int) : 'a =
    moi.[v]position <- (fun moi -> x')
end
```

Les classes peuvent être étendues par héritage. La classe `point_coloré` a les mêmes méthodes et variables d'instances que la classe `point`, ainsi qu'une méthode `couleur`.

```
class points_coloré (c0 : string) (x0 : int) = object
  inherit point x0
  method couleur : string = c0
end
```

Le type des objets de cette classe est :

```
Obj('a)[position: int; déplace: int -> 'a; couleur: string]
```

C'est un sous-type du type des objets de la classe `point`. Par conséquent, il est possible de considérer un point coloré comme un point.

5.1.2 Méthodes binaires

Étudions maintenant le cas d'une classe possédant une méthode binaire. La classe `comparable` a une méthode `x` retournant un entier ainsi qu'une méthode `égal` prenant un argument `autre` et vérifiant que l'appel de la méthode `x` retourne la même valeur pour `autre` et l'objet lui-même `moi`.

```
class comparable [protected comp] (x0 : int) = object (moi : 'a)
  method x : int = x0
  method égale (autre : 'a) : bool =
    moi.[comp]x = autre.[comp]x
end
```

Il ne serait pas correct de donner à un objet de cette classe le type

```
Obj('a)[x :int ; égale :'a -> bool]
```

En effet, l'argument de la méthode `égale` doit avoir une vue `comp`. Cette vue doit donc apparaître dans son type :

```
Obj('a)[x :int ;égale :'a -> bool | comp]
```

Cela est indiqué par le mot clé `protected` annotant la vue `comp`. Ce mot clé a deux effets. Tout d'abord, la vue reste visible après la définition de la classe, mais est abstraite (il n'est pas possible d'appeler des méthodes via cette vue en dehors du corps de la classe). De plus, la vue reste présente dans le type des objets de la classe. Par conséquent, la vue se comporte comme une marque indiquant la classe à laquelle un objet appartient. Quand aucun mot-clé n'est présent (comme dans les premières classes présentées), la vue est locale à la classe et n'apparaît pas dans le type des instances de la classe.

Il est possible de définir une sous-classe `comparable2` masquant la méthode `x` tout en laissant la méthode binaire `égale` visible. En effet, la méthode `égale` n'accède pas directement à la méthode `x` de son paramètre, mais utilise la vue `comp`. La méthode `x` est cachée en restreignant la signature de la classe par un type de classe.

```
class comparable2 [protected comp2] (x0 : int) :
  object ('a)
    method égale : 'a -> bool
  end = object inherit comparable x0 end
```

Le type d'un objet de la classe est :

```
Obj('a)[égale: 'a -> bool | comp2]
```

De nouveau, une vue apparaît dans le type. Ainsi, la méthode `égale` d'un objet de la classe `comparable2` ne peut être appliquée qu'à un objet qui possède également la vue `comp2`. Un tel objet doit être une instance de la classe `comparable2` ou d'une de ses sous-classes. Dans notre calcul, les vues sont toujours héritées (mais le type d'un objet peut ne montrer que certaines de ses vues). L'objet a donc également hérité de la vue `comp`, comme attendu par la définition de la méthode `égale`.

5.1.3 Vues publiques

Il est parfois pratique de pouvoir définir une vue dont le type reste visible au-delà de la définition de la classe qui l'introduit. Un exemple typique est celui des fonctions amies, c'est-à-dire des fonctions qui ont un accès privilégié aux objets d'une certaine classe (un exemple détaillé est donné dans la section 5.3.3). Le mot clé `public` est utilisé pour cela.

```
class point_coloré2 [public pointc] (c0 : string) (x0 : int) =
  object
    inherit point x0
    method couleur : string = c0
  end
```

Le type d'un objet de la classe `point_coloré2` est le même que celui d'un objet de la classe `point_coloré`, à part la présence de la vue `pointc`.

```
Obj('a)[couleur: string; get_x: int; move: int -> 'a | pointc]
```

Comme pour la classe `comparable`, la vue apparaît dans ce type. Cependant, la vue `pointc` est ici concrète, et il est donc possible d'appeler des méthodes via cette vue. Ainsi, il est possible de créer un objet de la classe `point_coloré2` et d'appeler la méthode `couleur` via la vue `pointc` plutôt que directement.

```
(new point_coloré2 "blue" 3).[pointc]couleur
```

Il serait également possible d'écrire une sous-classe `point_coloré2` masquant la méthode `couleur`, mais gardant la vue `pointc`. La méthode ne pourrait alors plus être appelée directement, mais il resterait possible de l'appeler via la vue `pointc`.

5.2 Calcul de base

Comme dans [13], le calcul distingue deux sortes d'objets par leur type : les *prototypes* et *« vrais objets »*. Des méthodes peuvent être ajoutées à un prototype, mais le sous-typage sur les prototypes est l'identité : masquer une méthode d'un prototype est une opération explicite. Au contraire, le sous-typage en largeur est autorisé sur les vrais objets, mais ils ne sont pas extensibles. Un prototype peut être transformé en objet par sous-typage.

Dans ce calcul, les prototypes peuvent avoir des méthodes abstraites. Ce n'est pas une nécessité et il serait facile de décrire une variante du calcul sans méthodes abstraites. Cependant, afin d'assurer la correction d'un calcul avec masquage de méthodes et ajout de méthodes, la redéfinition de méthodes et l'ajout de méthodes doivent être deux notions distinctes. En effet, dans le premier cas, les méthodes déjà existantes devront utiliser la nouvelle définition de la méthode, alors que dans le second cas, le comportement des méthodes déjà existantes doit être inchangé. Plutôt que d'utiliser ces deux opérations comme primitives comme dans [30], nous avons préféré décomposer l'ajout d'une méthode en l'ajout d'une méthode abstraite suivi de la redéfinition de cette méthode. Nous pensons que ces dernières primitives sont plus simples et plus orthogonales. De plus, elles permettent de définir des méthodes mutuellement récursives.

5.2.1 Syntaxe

La syntaxe du langage est présentée dans la figure 5.1. Une vue est un nom donnant accès à une collection de méthodes d'un objet. Les vues sont explicitement introduites : la construction $\rho(k : t)a$ définit une nouvelle vue k de type t, dont la portée est limitée à l'expression a. Cette construction est similaire à celle utilisée dans [33] pour représenter des emplacements mémoires (ce qui permet de formaliser les références), et à la construction ν du π-calcul [20]. Le type d'une vue $\zeta(\alpha)[\mathcal{L}]$ est le type des méthodes auxquelles elle donne accès : une fonction des noms de méthode vers les types, paramétrée par une variable de type α représentant l'objet auquel on accède via cette vue.

Une expression $\zeta(x : \alpha)[L : \mathcal{L}]^{\varphi}_{\Phi}$ définit un objet. La variable x représente l'objet lui-même et la variable de type α représente son type. La fonction L des noms de méthodes vers les expressions définit les méthodes de l'objet. Leurs types sont donnés par la fonction $\mathcal{L}$, des noms de méthodes vers les types. Une méthode peut être accédée soit directement (via *l'interface principale* de l'objet) soit par une vue (qui fournit une *interface alternative* à l'objet). Dans les deux cas, le nom externe de la méthode est traduit en son nom interne par un dictionnaire φ (un dictionnaire est une fonction injective des noms de méthodes vers les noms de méthodes). Par conséquent, un objet porte un dictionnaire φ pour son interface principale et une fonction Φ des vues vers les dictionnaires pour ses interfaces alternatives.

$a ::= x$	Variable
$\mid \lambda(x : \tau)a$	Abstraction
$\mid a(a')$	Application
$\mid \zeta(x : \alpha)[L : \mathcal{L}]^{\varphi}_{\Phi}$	Objet
$\mid a + (l : \varsigma(\alpha)\tau)$	Allocation d'une méthode
$\mid a \setminus l$	Masquage d'une méthode
$\mid a.l$	Sélection d'une méthode
$\mid a._k l$	Sélection d'une méthode via une vue
$\mid a.l \Leftarrow \varsigma(x : \alpha)a'$	Redéfinition d'une méthode
$\mid a._k l \Leftarrow \varsigma(x : \alpha)a'$	Redéfinition d'une méthode via une vue
$\mid \rho(k : t)a$	Introduction d'une vue
$\mid \langle a \rangle_k$	Ajout d'une vue
$\mid a\vert_k$	Remplacement d'une vue
$\mid \texttt{pack}\ a\ \texttt{as}\ \tau\ \texttt{hiding}\ \mathcal{K}$	Masquage
$\mid \texttt{open}\ a\ \texttt{as}\ [k, x]\ \texttt{in}\ a'$	Ouverture
$\tau ::= \alpha$	Variable de type
$\mid \tau \to \tau'$	Type fonctionnel
$\mid t_{\mathcal{K}}$	Type d'un objet
$\mid t^{\mathcal{A}}_{\mathcal{K}}$	Type de prototype
$\mid \exists(k)\tau$	Abstraction d'une vue
$t ::= \zeta(\alpha)[\mathcal{L}]$	Type d'une vue

FIG. 5.1: Syntaxe du calcul de base

Le type d'un objet $t_{\mathcal{K}}$ est composé du type de son interface principale t et d'une liste $\mathcal{K}$ de vues. Le type d'un prototype $t_{\mathcal{K}}^{\mathcal{A}}$ contient également la liste $\mathcal{A}$ des noms de ses méthodes abstraites.

La construction $a + (l : \varsigma(\alpha)\tau)$ ajoute une méthode abstraite l de type τ à l'objet a. La variable de type α représente le type de cet objet. Symétriquement, la construction $a \setminus l$ permet de masquer une méthode l. Ces constructions n'affectent que l'interface principale de l'objet. En effet, de telles constructions n'auraient pas de sens pour les autres interfaces car les vues ont un type fixe.

Les méthodes peuvent être appelées soit directement $(a.l)$ ou par une vue $(a._k l)$. L'alternative existe également pour la redéfinition d'une méthode : $a.l \Leftarrow \varsigma(x : \alpha)a'$ et $a._k l \Leftarrow \varsigma(x : \alpha)a'$. Il n'est possible de redéfinir une méthode que si elle apparaît déjà dans l'interface de l'objet. L'extension d'un objet se fait en lui ajoutant une méthode abstraite puis en la redéfinissant.

Une vue k peut être ajoutée à un objet $(\langle a \rangle_k)$ si elle a le même type que l'interface principal de l'objet. Il est également possible de remplacer l'interface principale d'un objet par l'interface d'une de ses vues en sélectionnant cette vue : $a|_k$.

Les opérations standards de masquage à l'aide de types existentiels [8] permettent d'abstraire les vues.

5.2.2 Sémantique dynamique

Nous avons défini une sémantique à réduction à petits pas pour notre calcul. La relation de réduction locale $\longrightarrow$ est définie dans la figure 5.2. Il est fait usage dans cette figure de plusieurs opérations de substitution, écrites $a\{v/x\}$, $a\{\tau/\alpha\}$ et $a\{\mathcal{K}/k\}$. Les deux premières sont les substitutions habituelles de valeurs et de types. La troisième substitue un ensemble de vues à une vue unique. Elle est telle que :

$$\begin{aligned} \mathcal{K}\{\mathcal{K}'/k\} &= \mathcal{K} \setminus \{k\} \cup \mathcal{K}' \text{ si } k \in \mathcal{K} \\ &= \mathcal{K} \text{ autrement.} \end{aligned}$$

L'ajout de méthodes modifie le type interne $\mathcal{L}$ de l'objet ainsi que son dictionnaire principal φ : une méthode fraîche l' est ajoutée au type $\mathcal{L}$ et le dictionnaire est étendu afin d'associer le nom de méthode interne l' au nom de méthode externe l. Le masquage d'une méthode n'affecte que le dictionnaire principal : le nom de la méthode est supprimé du dictionnaire ; ainsi, la méthode n'est plus accessible via ce dictionnaire. La redéfinition de méthode se fait en modifiant la liste des méthodes L. Elle peut se faire soit directement, soit via une vue k. Dans le premier cas, le nom de la méthode à redéfinir est traduit à l'aide du dictionnaire φ. Dans le second cas, le dictionnaire associé à la vue k, c'est-à-dire $\Phi(k)$ est utilisé. Pour l'appel d'une méthode l, la définition $\varsigma(x : \alpha)a$ de cette méthode est extraite de L. Puis la variable x est remplacée dans l'expression a par l'objet et la variable de type α est remplacée par un type convenable. La capture d'une vue k consiste à lier dans Φ la vue au dictionnaire courant φ. La sélection d'une vue k consiste à remplacer le dictionnaire principal par le dictionnaire $\Phi(k)$ associé à la vue. Pour l'ouverture d'une existentielle, la vue k est remplacée par sa valeur effective, c'est-à-dire l'ensemble de vues $\mathcal{K}$, et la variable x est remplacée par le contenu v' de l'existentielle. La dernière règle de réduction permet de pousser l'introduction d'une vue vers l'extérieur (extrusion).

Allocation d'une méthode ($l' \notin \operatorname{dom} \mathcal{L} \cup \operatorname{img} \varphi \cup \bigcup_{k \in \operatorname{dom} \Phi} \operatorname{img} \Phi(k)$)

$$\zeta(x : \alpha)[L : \mathcal{L}]^{\varphi}_{\Phi} + (l : \varsigma(\alpha)\tau) \quad \longrightarrow \quad \zeta(x : \alpha)[L : (\mathcal{L}; (l' : \tau))]^{\varphi;(l=l')}_{\Phi}$$

Masquage d'une méthode

$$\zeta(x : \alpha)[L : \mathcal{L}]^{\varphi}_{\Phi} \setminus l \quad \longrightarrow \quad \zeta(x : \alpha)[L : \mathcal{L}]^{\varphi|_{\operatorname{dom}\varphi \setminus \{l\}}}_{\Phi}$$

Redéfinition d'une méthode

$$\zeta(x : \alpha)[L : \mathcal{L}]^{\varphi}_{\Phi}.l \Leftarrow \varsigma(x : \alpha)a \quad \longrightarrow \quad \zeta(x : \alpha)[L; (\varphi(l) = a) : \mathcal{L}]^{\varphi}_{\Phi}$$
$$\zeta(x : \alpha)[L : \mathcal{L}]^{\varphi}_{\Phi}._k l \Leftarrow \varsigma(x : \alpha)a \quad \longrightarrow \quad \zeta(x : \alpha)[L; (\Phi(k)(l) = a) : \mathcal{L}]^{\varphi}_{\Phi}$$

Sélection d'une méthode ($v = \zeta(x : \alpha)[L : \mathcal{L}]^{\varphi}_{\Phi}$ et $\tau = \zeta(\alpha)[\mathcal{L} \circ \varphi]_{\operatorname{dom}\Phi}$)

$$v.l \quad \longrightarrow \quad L(\varphi(l))\{\tau/\alpha\}\{v/x\}$$
$$v._k l \quad \longrightarrow \quad L(\Phi(k)(l))\{\tau/\alpha\}\{v/x\}$$

Manipulation d'une vue

$$\langle \zeta(x : \alpha)[L : \mathcal{L}]^{\varphi}_{\Phi} \rangle_k \quad \longrightarrow \quad \zeta(x : \alpha)[L : \mathcal{L}]^{\varphi}_{\Phi;(k=\varphi)}$$
$$\zeta(x : \alpha)[L : \mathcal{L}]^{\varphi}_{\Phi}|_k \quad \longrightarrow \quad \zeta(x : \alpha)[L : \mathcal{L}]^{\Phi(k)}_{\Phi}$$

Application

$$(\lambda(x : \tau)a)(v) \quad \longrightarrow \quad a\{v/x\}$$

Ouverture ($v = \texttt{pack}\ v'\ \texttt{as}\ \exists(k)\tau\ \texttt{hiding}\ \mathcal{K}$)

$$\texttt{open}\ v\ \texttt{as}\ [k, x]\ \texttt{in}\ a \quad \longrightarrow \quad a\{\mathcal{K}/k\}\{v'/x\}$$

Introduction d'une vue ($F \neq [\_]$)

$$F[\rho(k : t)a] \quad \longrightarrow \quad \rho(k : t)F[a]$$

FIG. 5.2: Règles de réduction

Deux sortes de contextes d'évaluation, E et F, sont utilisés pour définir la sémantique :

$$\begin{array}{ll} E ::= [\_] & F ::= [\_] \\ \quad \mid \rho(k:t)E & \quad \mid F + (l : \varsigma(\alpha)\tau) \\ & \quad \mid F.l \Leftarrow \varsigma(x:\alpha)a \\ & \quad \mid F.{}_k l \Leftarrow \varsigma(x:\alpha)a \\ & \quad \mid F.l \\ & \quad \mid F.{}_k l \\ & \quad \mid \langle F \rangle_k \\ & \quad \mid F|_k \\ & \quad \mid F(a) \\ & \quad \mid v(F) \\ & \quad \mid \texttt{pack}\ F\ \texttt{as}\ \tau\ \texttt{hiding}\ \mathcal{K} \\ & \quad \mid \texttt{open}\ F\ \texttt{as}\ [k,x]\ \texttt{in}\ a \end{array}$$

La relation de réduction locale $\longrightarrow$ est étendue en une relation définissant un pas d'évaluation : $a \longrightarrow a'$ si et seulement si il existe des expressions a_1 et a'_1 telles que $a = E[F[a_1]]$, $a_1 \longrightarrow a'_1$ et $a' = E[F[a'_1]]$.

Les valeurs sont définies par la sous-grammaire des expressions suivante :

$$\begin{array}{lr} v ::= \lambda(x:\tau)a & \text{Abstraction} \\ \quad \mid \zeta(x:\alpha)[L:\mathcal{L}]^{\varphi}_{\Phi} & \text{Object} \\ \quad \mid \texttt{pack}\ v\ \texttt{as}\ \tau\ \texttt{hiding}\ \mathcal{K} & \text{Abstraction de vue} \end{array}$$

La sémantique que nous donnons n'est pas déterministe. En effet, lorsqu'une méthode est ajoutée à un prototype, son nom interne peut être choisi arbitrairement. Cette sémantique pourrait être rendue déterministe en utilisant une fonction de choix associant un nouveau nom de méthode interne à l'ensemble des noms des méthodes déjà présentes (c'est le choix fait dans [30]). On peut également remarquer que les noms de méthodes internes ne sont pas observables. Il est donc possible d'identifier les objets et les prototypes modulo renommage de leurs méthodes internes.

5.2.3 Système de type

Un environnement contient des liaisons de variables et de vues. Il contient également des hypothèses de filtrage, c'est-à-dire l'hypothèse qu'un type α est un type d'objet possédant au moins un certain ensemble de vues $\mathcal{K}$ (ce n'est pas la même notion de filtrage que celle introduite par Bruce [6], mais les deux notions sont similaires : les types filtrés par un certain ensemble de vues ont une forme similaire, sans être nécessairement sous-types d'un même type).

$$\begin{array}{lr} \Gamma ::= \emptyset & \text{Environnement vide} \\ \quad \mid \Gamma; (x:\tau) & \text{Liaison de valeur} \\ \quad \mid \Gamma; (\alpha <\# \mathcal{K}) & \text{Hypothèse de filtrage} \\ \quad \mid \Gamma; (k:t) & \text{Liaison de vue} \end{array}$$

$\mid \Gamma; (k)$ Liaison de vue abstraite

Les règles de typage font usage de trois sortes de jugements : des jugements de typage $\Gamma \vdash a : \tau$, des jugements de sous-typage $\Gamma \vdash \tau \preceq \tau'$ et des jugements de filtrage $\Gamma \vdash \alpha <\# \mathcal{K}$.

Un type τ, un type de vue t, une expression a sont *fermés* par rapport à un environnement Γ si toutes leurs variables sont liées dans l'environnement. Un environnement Γ est fermé si tous les types et les types de vues de son image sont fermés par rapport à Γ. Un jugement de typage $\Gamma \vdash a : \tau$ est fermé si Γ est fermé et a et τ sont fermés par rapport à Γ ; un jugement de sous-typage $\Gamma \vdash \tau \preceq \tau'$ est fermé si Γ est fermé et τ et τ' sont fermés par rapport à Γ. On fera l'hypothèse que tous les jugements considérés sont fermés, et qu'aucune variable n'est liée deux fois dans un environnement.

Certaines règles de typage nécessitent qu'une variable de type α apparaisse en position covariante dans un type τ (ce qui sera noté $\mathrm{co}_\alpha(\tau)$). Une variable de type α apparaît en position covariante dans un type τ si l'une des conditions suivantes est vraie :

- α n'est pas libre dans τ ;
- τ est α ;
- τ est de la forme $\tau_1 \to \tau_2$ et α apparaît en position contravariante dans τ_1 et en position covariante dans τ_2 ;
- τ est de la forme $\exists(k)\tau'$ et α apparaît en position covariante dans τ'.

De même, une variable de type α apparaît en position contravariante dans un type τ si l'une des conditions suivantes est vraie :

- α n'est pas libre dans τ ;
- τ est de la forme $\tau_1 \to \tau_2$ et α apparaît en position covariante dans τ_1 et en position contravariante dans τ_2 ;
- τ est de la forme $\exists(k)\tau'$ et α apparaît en position contravariante dans τ'.

Les règles de typage sont présentées dans les figures 5.3, 5.4 et 5.5. Nous ne décrivons que les principales règles de typage. La règle la plus complexe est la règle Proto. Elle commence par définir les différents composants du type du prototype. Le type $\mathcal{L}'$ de l'interface principale d'un prototype est obtenu par traduction du type des méthodes du prototype par le dictionnaire φ. Les méthodes abstraites $\mathcal{A}$ sont les méthodes de $\mathcal{L}'$ qui ne sont pas définies, c'est-à-dire qui n'apparaissent pas dans la liste des méthodes L du prototype. Les vues $\mathcal{K}$ attachées au prototype sont les vues pour lesquelles la fonction Φ définit un dictionnaire. La condition $\mathrm{dom}\,\mathcal{L} \subseteq \mathrm{dom}\,L \cup \mathrm{img}\,\varphi$ énonce que les méthodes de $\mathcal{L}$ soit sont des méthodes définies du prototype, soit sont visibles de l'extérieur. La condition suivante est que $\mathcal{L}$ doit donner le type des méthodes de L. Enfin, la dernière condition énonce que les vues du prototype ne doivent pas être abstraites, et que le type d'une vue k correspond à la traduction de $\mathcal{L}$ par le dictionnaire $\Phi(k)$ associé à cette vue. Le type de l'expression $a + (l : \varsigma(\alpha)\tau)$ est le type de l'expression a où la méthode l a été ajoutée à la vue principale $\mathcal{L}$ et à l'ensemble $\mathcal{A}$ des méthodes abstraites (règle Extension). Le type de l'expression $a \setminus l$ est le type de l'expression a où la méthode l a été supprimée de la vue principale $\mathcal{L}$, pourvu que cette méthode ne soit pas abstraite (règle Restriction). Une vue peut être capturée par un prototype (règle Vue-Capture) si son type est

(Var)
$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}$$

(Abs)
$$\frac{\Gamma; (x : \tau') \vdash a : \tau}{\Gamma \vdash \lambda(x : \tau')a : \tau' \to \tau}$$

(App)
$$\frac{\Gamma \vdash a : \tau' \to \tau \qquad \Gamma \vdash a' : \tau'}{\Gamma \vdash a(a') : \tau}$$

(Sub)
$$\frac{\Gamma \vdash a : \tau \qquad \Gamma \vdash \tau \preceq \tau'}{\Gamma \vdash a : \tau'}$$

(Proto)
$$\frac{\begin{array}{ll} \mathcal{L}' = \mathcal{L} \circ \varphi & \mathcal{A} = \operatorname{dom} \mathcal{L}' \setminus \operatorname{dom}(L \circ \varphi) \\ \mathcal{K} = \operatorname{dom} \Phi & \operatorname{dom} \mathcal{L} \subseteq \operatorname{dom} L \cup \operatorname{img} \varphi \\ \multicolumn{2}{l}{\text{Pour tout } l \text{ dans } \operatorname{dom} L,\ \Gamma; (\alpha <\# \mathcal{K}); (x : \alpha) \vdash L(l) : \mathcal{L}(l)} \\ \multicolumn{2}{l}{\forall k \in \operatorname{dom} \Phi \cdot (k : \zeta(\alpha)[\mathcal{L}'']) \in \Gamma \wedge \mathcal{L}'' = \mathcal{L} \circ \Phi(k)} \end{array}}{\Gamma \vdash \zeta(x : \alpha)[L : \mathcal{L}]^{\varphi}_{\Phi} : \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}}}$$

(Extension)
$$\frac{\Gamma \vdash a : \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}}{\Gamma \vdash a + (l : \varsigma(\alpha)\tau) : \zeta(\alpha)[\mathcal{L}; (l : \tau)]^{\mathcal{A} \cup \{l\}}_{\mathcal{K}}}$$

(Restriction)
$$\frac{\Gamma \vdash a : \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}} \qquad l \in \operatorname{dom} \mathcal{L} \setminus \mathcal{A}}{\Gamma \vdash a \setminus l : \zeta(\alpha)[\mathcal{L}|_{\operatorname{dom} \mathcal{L} \setminus \{l\}}]^{\mathcal{A}}_{\mathcal{K}}}$$

(Sélection)
$$\frac{\Gamma \vdash a : \tau \qquad \tau = \zeta(\alpha)[\mathcal{L}]_{\mathcal{K}}}{\Gamma \vdash a.l : (\mathcal{L}(l))\{\tau/\alpha\}}$$

(Vue-Sélection)
$$\frac{\begin{array}{c} \Gamma \vdash a : \tau \qquad \Gamma \vdash \tau <\# \mathcal{K} \\ k \in \mathcal{K} \qquad (k : \zeta(\alpha)[\mathcal{L}']) \in \Gamma \end{array}}{\Gamma \vdash a._k l : (\mathcal{L}'(l))\{\tau/\alpha\}}$$

(Redéfinition)
$$\frac{\begin{array}{c} \Gamma \vdash a : \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}} \\ \Gamma; (\alpha <\# \mathcal{K}); (x : \alpha) \vdash a' : \mathcal{L}(l) \end{array}}{\Gamma \vdash a.l \Leftarrow \varsigma(x : \alpha)a' : \zeta(\alpha)[\mathcal{L}]^{\mathcal{A} \setminus \{l\}}_{\mathcal{K}}}$$

(Vue-Redéfinition)
$$\frac{\begin{array}{c} \Gamma \vdash a : \tau \qquad \Gamma \vdash \tau <\# \mathcal{K} \\ k \in \mathcal{K} \qquad (k : \zeta(\alpha)[\mathcal{L}']) \in \Gamma \\ \Gamma; (\alpha <\# \mathcal{K}); (x : \alpha) \vdash a' : \mathcal{L}'(l) \end{array}}{\Gamma \vdash a._k l \Leftarrow \varsigma(x : \alpha)a' : \tau}$$

(Vue-Abs)
$$\frac{\Gamma; (k : t) \vdash a : \tau}{\Gamma \vdash \rho(k : t)a : \tau}$$

(Vue-Capture)
$$\frac{\begin{array}{c} \Gamma \vdash a : \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}} \\ (k : \zeta(\alpha)[\mathcal{L}]) \in \Gamma \end{array}}{\Gamma \vdash \langle a \rangle_k : \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K} \cup \{k\}}}$$

(Vue-Remplace)
$$\frac{\begin{array}{c} \Gamma \vdash a : \zeta(\alpha)[\mathcal{L}]_{\mathcal{K}} \\ (k : \zeta(\alpha)[\mathcal{L}']) \in \Gamma \\ k \in \mathcal{K} \end{array}}{\Gamma \vdash a|_k : \zeta(\alpha)[\mathcal{L}']_{\mathcal{K}}}$$

(Pack)
$$\frac{\Gamma \vdash a : \tau\{\mathcal{K}/k\}}{\Gamma \vdash \texttt{pack}\ a\ \texttt{as}\ \exists(k)\tau\ \texttt{hiding}\ \mathcal{K} : \exists(k)\tau}$$

(Open)
$$\frac{\begin{array}{c} \Gamma \vdash a' : \exists(k)\tau' \\ \Gamma; (k); (x : \tau') \vdash a : \tau \end{array}}{\Gamma \vdash \texttt{open}\ a'\ \texttt{as}\ [k, x]\ \texttt{in}\ a : \tau}$$

Fig. 5.3: Règles de typage des expressions

(Sub-Réfl)
$$\frac{}{\Gamma \vdash \tau \preceq \tau}$$

(Sub-Flèche)
$$\frac{\Gamma \vdash \tau_1 \preceq \tau_2 \qquad \Gamma \vdash \tau_2' \preceq \tau_1'}{\Gamma \vdash \tau_1' \rightarrow \tau_1 \preceq \tau_2' \rightarrow \tau_2}$$

(Sub-Proto)
$$\frac{\Gamma \vdash \zeta(\alpha)[\mathcal{L}]_{\mathcal{K}} \preceq \zeta(\alpha)[\mathcal{L}']_{\mathcal{K}'}}{\Gamma \vdash \zeta(\alpha)[\mathcal{L}]^{\emptyset}_{\mathcal{K}} \preceq \zeta(\alpha)[\mathcal{L}']_{\mathcal{K}'}}$$

(Sub-Objet)
$$\frac{\mathcal{L}' \subseteq \mathcal{L} \qquad \mathrm{co}_\alpha(\mathcal{L}') \qquad \mathcal{K}' \subseteq \mathcal{K} \\ \forall k \in \mathcal{K}' \cdot (k : \zeta(\alpha)[\mathcal{L}'']) \in \Gamma \wedge \mathrm{co}_\alpha(\mathcal{L}'')}{\Gamma \vdash \zeta(\alpha)[\mathcal{L}]_{\mathcal{K}} \preceq \zeta(\alpha)[\mathcal{L}']_{\mathcal{K}'}}$$

(Sub-Filtrage)
$$\frac{\Gamma \vdash \alpha <\# \mathcal{K} \\ \forall k \in \mathcal{K} \cdot (k : \zeta(\alpha)[\mathcal{L}'']) \in \Gamma \wedge \mathrm{co}_\alpha(\mathcal{L}'')}{\Gamma \vdash \alpha \preceq \zeta(\alpha')[\emptyset]_{\mathcal{K}}}$$

Fig. 5.4: Règles de sous-typage

(Filtrage-Var)
$$\frac{(\alpha <\# \mathcal{K}) \in \Gamma \qquad \mathcal{K}' \subseteq \mathcal{K}}{\Gamma \vdash \alpha <\# \mathcal{K}'}$$

(Filtrage-Obj)
$$\frac{\mathcal{K}' \subseteq \mathcal{K}}{\Gamma \vdash \zeta(\alpha)[\mathcal{L}]_{\mathcal{K}} <\# \mathcal{K}'}$$

Fig. 5.5: Règles de filtrage

le même que celui de la vue principale du prototype. La vue est ajoutée à l'ensemble des vues du type du prototype. Une méthode l peut être appelée via une vue k (règle SÉLECTION) si cette vue est l'une des vues de l'objet et si elle n'est pas abstraite. Le type du résultat est le type de la méthode dans cette vue.

5.2.4 Correction

Nous avons montré la préservation du typage par réduction et l'absence de bloquage.

Préservation du typage par réduction

Théorème 17 (Préservation du typage par réduction) *Si* $\Gamma \vdash a : \tau$ *et* $a \longrightarrow a'$, *alors* $\Gamma \vdash a' : \tau$.

La preuve suivante utilise un certain nombre de lemmes qui sont présentés plus loin.

Démonstration. Par cas sur la règle de réduction utilisée.

Grâce à la transitivité (lemme 24) et à la réflexivité du sous-typage, on peut supposer sans perte de généralité que, dans la preuve d'un jugement $\Gamma \vdash a : \tau$, l'usage de la règle de sous-typage SUB alterne avec l'usage d'une autre règle de typage. Il est également suffisant de ne considérer que les cas où la preuve ne se termine pas pas l'utilisation du sous-typage. Enfin, d'après le lemme 18, il suffit de considérer les règles de réduction locale.

– $\zeta(x : \alpha)[L : \mathcal{L}]^{\varphi}_{\Phi} + (l : \varsigma(\alpha)\tau) \longrightarrow \zeta(x : \alpha)[L : (\mathcal{L}; (l' : \tau))]^{\varphi;(l=l')}_{\Phi}$ où $l' \notin \operatorname{dom}\mathcal{L} \cup \operatorname{img}\varphi$.

La preuve de la première hypothèse doit se terminer par l'usage de la règle EXTENSION. On a donc $\Gamma \vdash \zeta(x : \alpha)[L : \mathcal{L}]^{\varphi}_{\Phi} : \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}}$ pour des $\mathcal{L}'$, $\mathcal{A}$ et $\mathcal{K}$ tels que $\tau = \zeta(\alpha)[\mathcal{L}'; (l : \tau)]^{\mathcal{A}\cup\{l\}}_{\mathcal{K}}$. Ce jugement doit être prouver à l'aide de la règle PROTO. Donc,

$$\left\{\begin{array}{l} \mathcal{L}' = \mathcal{L} \circ \varphi \\ \mathcal{A} = \operatorname{dom}\mathcal{L}' \setminus \operatorname{dom}(L \circ \varphi) \\ \mathcal{K} = \operatorname{dom}\Phi \\ \operatorname{dom}\mathcal{L} \subseteq \operatorname{dom} L \cup \operatorname{img}\varphi \\ \text{Quel que soit } l \text{ dans } \operatorname{dom} L,\ \Gamma; (\alpha <\# \mathcal{K}); (x : \alpha) \vdash L(l) : \mathcal{L}(l) \\ \forall k \in \operatorname{dom}\Phi \cdot (k : \zeta(\alpha)[\mathcal{L}'']) \in \Gamma \wedge \mathcal{L}'' = \mathcal{L} \circ \Phi(k) \end{array}\right.$$

Soient $\varphi_0 = (\varphi; (l = l'))$, $\mathcal{L}_0 = (\mathcal{L}; (l' = \tau))$, $\mathcal{L}'_0 = (\mathcal{L}'; (l = \tau))$ et $\mathcal{A}_0 = \mathcal{A} \cup \{l\}$. Nous allons prouver que les hypothèses ci-dessus sont toujours vérifiées après la substitution de φ par φ_0, de $\mathcal{L}$ par $\mathcal{L}_0$, de $\mathcal{L}'$ par $\mathcal{L}'_0$, et de $\mathcal{A}$ par $\mathcal{A}_0$. Cela terminera la preuve de ce cas, car cela entraîne $\Gamma \vdash \zeta(x : \alpha)[L : \mathcal{L}_0]^{\varphi_0}_{\Phi} : \zeta(\alpha)[\mathcal{L}'_0]^{\mathcal{A}_0}_{\mathcal{K}}$, c'est-à-dire, $\Gamma \vdash a' : \tau$.

$$\begin{array}{rcll} \mathcal{L}_0 \circ \varphi_0 & = & (\mathcal{L}; (l' = \tau)) \circ (\varphi; (l = l')) & \\ & = & ((\mathcal{L}; (l' = \tau)) \circ \varphi); (l = \tau) & \\ & = & (\mathcal{L} \circ \varphi); (l = \tau) & (l' \notin \operatorname{dom}\mathcal{L} \cup \operatorname{img}\varphi) \\ & = & \mathcal{L}'; (l = \tau) & \\ & = & \mathcal{L}'_0 & \end{array}$$

$$\begin{array}{rcl}\operatorname{dom}\mathcal{L}'_0 \setminus \operatorname{dom}(L \circ \varphi_0) & = & \operatorname{dom}(\mathcal{L}'; (l = \tau)) \setminus \operatorname{dom}(L \circ (\varphi; (l = l'))) \\ & = & (\operatorname{dom}\mathcal{L}' \cup \{l\}) \setminus \operatorname{dom}(L \circ (\varphi; (l = l'))) \\ & = & (\operatorname{dom}\mathcal{L}' \cup \{l\}) \setminus (\operatorname{dom}(L \circ \varphi) \setminus \{l\}) \\ & & \qquad (l' \notin \operatorname{dom}\mathcal{L} \text{ et } \operatorname{dom} L \subseteq \operatorname{dom}\mathcal{L}) \\ & = & (\operatorname{dom}\mathcal{L}' \setminus \operatorname{dom}(L \circ \varphi)) \cup \{l\} \\ & = & \mathcal{A} \cup \{l\} \\ & = & \mathcal{A}_0 \end{array}$$

$$\begin{array}{rcl}\operatorname{dom} L \cup \operatorname{img}\varphi_0 & = & \operatorname{dom} L \cup \operatorname{img}(\varphi; (l = l')) \\ & = & \operatorname{dom} L \cup \operatorname{img}\varphi \cup \{l'\} \\ & \supseteq & \operatorname{dom}\mathcal{L} \cup \{l'\} \\ & \supseteq & \operatorname{dom}\mathcal{L}_0 \end{array}$$

Finalement, comme $l' \notin \operatorname{dom}\mathcal{L}$, on a $\mathcal{L} \subseteq \mathcal{L}_0$. Ainsi, les deux dernières prémisses sont encore vérifiées après substitution.

– $\zeta(x : \alpha)[L : \mathcal{L}]^{\varphi}_{\Phi} \setminus l \longrightarrow \zeta(x : \alpha)[L : \mathcal{L}]^{\varphi|_{\operatorname{dom}\varphi\setminus\{l\}}}_{\Phi}$
La preuve de la première hypothèse doit se terminer par l'usage de la règle Restriction. On a donc $\Gamma \vdash \zeta(x : \alpha)[L : \mathcal{L}]^{\varphi}_{\Phi} : \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}}$ pour des $\mathcal{L}'$, $\mathcal{A}$ et $\mathcal{K}$ tels que $\tau = \zeta(\alpha)[\mathcal{L}'|_{\operatorname{dom}\mathcal{L}'\setminus\{l\}}]^{\mathcal{A}}_{\mathcal{K}}$ et $l \in \operatorname{dom}\mathcal{L}' \setminus \mathcal{A}$. Ce jugement doit être prouver à l'aide de la règle Proto. Donc,

$$\left\{\begin{array}{l}\mathcal{L}' = \mathcal{L} \circ \varphi \\ \mathcal{A} = \operatorname{dom}\mathcal{L}' \setminus \operatorname{dom}(L \circ \varphi) \\ \mathcal{K} = \operatorname{dom}\Phi \\ \operatorname{dom}\mathcal{L} \subseteq \operatorname{dom} L \cup \operatorname{img}\varphi \\ \text{Quel que soit } l \text{ dans } \operatorname{dom} L,\ \Gamma; (\alpha <\# \mathcal{K}); (x : \alpha) \vdash L(l) : \mathcal{L}(l) \\ \forall k \in \operatorname{dom}\Phi \cdot (k : \zeta(\alpha)[\mathcal{L}'']) \in \Gamma \wedge \mathcal{L}'' = \mathcal{L} \circ \Phi(k) \end{array}\right.$$

Soient $\varphi_0 = \varphi|_{\operatorname{dom}\varphi\setminus\{l\}}$ et $\mathcal{L}'_0 = \mathcal{L}'|_{\operatorname{dom}\mathcal{L}'\setminus\{l\}}$. Nous allons prouver que les hypothèses ci-dessus sont toujours vérifiées après la substitution de φ par φ_0 et $\mathcal{L}'$ par $\mathcal{L}'_0$. Cela terminera la preuve de ce cas, car cela entraîne $\Gamma \vdash \zeta(x : \alpha)[L : \mathcal{L}]^{\varphi_0}_{\Phi} : \zeta(\alpha)[\mathcal{L}'_0]^{\mathcal{A}}_{\mathcal{K}}$, c'est-à-dire, $\Gamma \vdash a' : \tau$.

$$\begin{array}{rcl}\mathcal{L} \circ \varphi_0 & = & \mathcal{L} \circ (\varphi|_{\operatorname{dom}\varphi\setminus\{l\}}) \\ & = & (\mathcal{L} \circ \varphi)|_{\operatorname{dom}(\mathcal{L}\circ\varphi)\setminus\{l\}} \\ & = & \mathcal{L}'|_{\operatorname{dom}\mathcal{L}'\setminus\{l\}} \\ & = & \mathcal{L}'_0 \end{array}$$

$$\begin{array}{rcl}\operatorname{dom}\mathcal{L}'_0 \setminus \operatorname{dom}(L \circ \varphi_0) & = & \operatorname{dom}(\mathcal{L}'|_{\operatorname{dom}\mathcal{L}'\setminus\{l\}}) \setminus \operatorname{dom}(L \circ (\varphi|_{\operatorname{dom}\varphi\setminus\{l\}})) \\ & = & (\operatorname{dom}\mathcal{L}' \setminus \{l\}) \setminus \operatorname{dom}((L \circ \varphi)|_{\operatorname{dom}(L\circ\varphi)\setminus\{l\}}) \\ & = & (\operatorname{dom}\mathcal{L}' \setminus \{l\}) \setminus (\operatorname{dom}(L \circ \varphi) \setminus \{l\}) \\ & = & (\operatorname{dom}\mathcal{L}' \setminus \operatorname{dom}(L \circ \varphi)) \setminus \{l\} \\ & = & \mathcal{A} \setminus \{l\} \\ & = & \mathcal{A} \qquad\qquad (l \in \operatorname{dom}\mathcal{L}' \setminus \mathcal{A}) \end{array}$$

Pour la prémisse suivante, nous allons prouver que $\varphi(l) \in \operatorname{dom} L$. Pour cela, nous devons prouver que $l \in \operatorname{dom}(L \circ \varphi)$. On sait que $l \in \operatorname{dom} \mathcal{L}' \setminus \mathcal{A}$. D'autre part :

$$\begin{array}{rcl} \operatorname{dom} \mathcal{L}' \setminus \mathcal{A} & = & \operatorname{dom} \mathcal{L}' \setminus (\operatorname{dom} \mathcal{L}' \setminus \operatorname{dom}(L \circ \varphi)) \\ & = & \operatorname{dom} \mathcal{L}' \cap \operatorname{dom}(L \circ \varphi) \\ & \subseteq & \operatorname{dom}(L \circ \varphi) \end{array}$$

Alors :

$$\begin{array}{rcl} \operatorname{dom} L \cup \operatorname{img} \varphi_0 & = & \operatorname{dom} L \cup \operatorname{img}(\varphi|_{\operatorname{dom} \varphi \setminus \{l\}}) \\ & = & (\operatorname{dom} L \cup \{\varphi(l)\}) \cup \operatorname{img}(\varphi|_{\operatorname{dom} \varphi \setminus \{l\}}) \\ & & \qquad (l \in \operatorname{dom}(L \circ \varphi)) \\ & = & \operatorname{dom} L \cup \operatorname{img} \varphi \\ & \supseteq & \operatorname{dom} \mathcal{L} \end{array}$$

Les autres prémisses sont inchangées.

- $\zeta(x:\alpha)[L:\mathcal{L}]^{\varphi}_{\Phi}.l \Leftarrow \varsigma(x:\alpha)a'' \longrightarrow \zeta(x:\alpha)[L;(\varphi(l)=a''):\mathcal{L}]^{\varphi}_{\Phi}$
  La preuve de la première hypothèse doit se terminer par l'usage de la règle REDÉFINITION. On a donc $\Gamma \vdash \zeta(x:\alpha)[L:\mathcal{L}]^{\varphi}_{\Phi} : \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}}$ pour des $\mathcal{L}'$, $\mathcal{A}$ et $\mathcal{K}$ tels que $\tau = \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}\setminus\{l\}}_{\mathcal{K}}$. De plus, $\Gamma;(\alpha <\# \mathcal{K});(x:\alpha) \vdash a'' : \mathcal{L}'(l)$. La règle PROTO doit être utilisée pour prouver le premier jugement. On a donc

$$\left\{ \begin{array}{l} \mathcal{L}' = \mathcal{L} \circ \varphi \\ \mathcal{A} = \operatorname{dom} \mathcal{L}' \setminus \operatorname{dom}(L \circ \varphi) \\ \mathcal{K} = \operatorname{dom} \Phi \\ \operatorname{dom} \mathcal{L} \subseteq \operatorname{dom} L \cup \operatorname{img} \varphi \\ \text{Quel que soit } l' \text{ dans } \operatorname{dom} L,\ \Gamma;(\alpha <\# \mathcal{K});(x:\alpha) \vdash L(l') : \mathcal{L}(l') \\ \forall k \in \operatorname{dom} \Phi \cdot (k : \zeta(\alpha)[\mathcal{L}'']) \in \Gamma \wedge \mathcal{L}'' = \mathcal{L} \circ \Phi(k) \end{array} \right.$$

Soient $L_0 = (L;(\varphi(l)=a''))$, $\mathcal{A}_0 = \mathcal{A} \setminus \{l\}$. Nous allons prouver que les hypothèses ci-dessus sont toujours vérifiées après la substitution de L par L_0 et $\mathcal{A}$ par $\mathcal{A}_0$. Cela terminera la preuve de ce cas, car cela entraîne $\Gamma \vdash \zeta(x:\alpha)[L_0:\mathcal{L}]^{\varphi}_{\Phi} : \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}_0}_{\mathcal{K}}$, c'est-à-dire, $\Gamma \vdash a' : \tau$.

$$\begin{array}{rcl} \operatorname{dom} \mathcal{L}' \setminus \operatorname{dom}(L_0 \circ \varphi) & = & \operatorname{dom} \mathcal{L}' \setminus \operatorname{dom}((L;(\varphi(l)=a'')) \circ \varphi) \\ & = & \operatorname{dom} \mathcal{L}' \setminus (\operatorname{dom}(L \circ \varphi) \cup \{l\}) \\ & = & (\operatorname{dom} \mathcal{L}' \setminus \operatorname{dom}(L \circ \varphi)) \setminus \{l\}) \\ & = & \mathcal{A} \setminus \{l\}) \\ & = & \mathcal{A}_0 \end{array}$$

$$\begin{array}{rcl} \operatorname{dom} L_0 \cup \operatorname{img} \varphi & = & \operatorname{dom}(L;(\varphi(l)=a'')) \cup \operatorname{img} \varphi \\ & = & \operatorname{dom} L \cup \{\varphi(l)\} \cup \operatorname{img} \varphi \\ & = & \operatorname{dom} L \cup \operatorname{img} \varphi \\ & \supseteq & \operatorname{dom} \mathcal{L} \end{array}$$

Soit $l' \in \operatorname{dom} L_0$. Si l'on suppose que $l' \neq \varphi(l)$, alors $L_0(l') = L(l')$. D'où, $\Gamma; (\alpha <\# \mathcal{K}); (x : \alpha) \vdash L_0(l') : \mathcal{L}(l')$. Supposons maintenant $l' = \varphi(l)$. Alors $L_0(l') = a''$. Mais $\Gamma; (\alpha <\# \mathcal{K}); (x : \alpha) \vdash a'' : \mathcal{L}'(l)$ et $\mathcal{L}'(l) = (\mathcal{L} \circ \varphi)(l) = \mathcal{L}(l')$. D'où, encore, $\Gamma; (\alpha <\# \mathcal{K}); (x : \alpha) \vdash L_0(l') : \mathcal{L}(l')$.

Les autres prémisses sont inchangées.

– $\zeta(x : \alpha)[L : \mathcal{L}]^{\varphi}_{\Phi} \cdot_k l \Leftarrow \varsigma(x : \alpha)a'' \longrightarrow \zeta(x : \alpha)[L; (\Phi(k)(l) = a'') : \mathcal{L}]^{\varphi}_{\Phi}$
La preuve de la première hypothèse doit se terminer par l'usage de la règle Vue-Redéfinition. Par inspection des règles de typage, cette règle est précédée de sous-typage et de la règle Proto. On a donc

$$\begin{cases} \tau = \zeta(\alpha)[\mathcal{L}_1]_{\mathcal{K}'} \\ \Gamma \vdash \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}} \preceq \tau \\ \mathcal{L}' = \mathcal{L} \circ \varphi \\ \mathcal{A} = \operatorname{dom} \mathcal{L}' \setminus \operatorname{dom}(L \circ \varphi) \\ \mathcal{K} = \operatorname{dom} \Phi \\ \operatorname{dom} \mathcal{L} \subseteq \operatorname{dom} L \cup \operatorname{img} \varphi \\ \text{Quel que soit } l' \text{ dans } \operatorname{dom} L,\ \Gamma; (\alpha <\# \mathcal{K}); (x : \alpha) \vdash L(l') : \mathcal{L}(l') \\ \forall k \in \operatorname{dom} \Phi \cdot (k : \zeta(\alpha)[\mathcal{L}'']) \in \Gamma \wedge \mathcal{L}'' = \mathcal{L} \circ \Phi(k) \\ \Gamma \vdash \tau <\# \mathcal{K}'' \\ k \in \mathcal{K}'' \\ (k : \zeta(\alpha)[\mathcal{L}'']) \in \Gamma \\ \Gamma; (\alpha <\# \mathcal{K}''); (x : \alpha) \vdash a'' : \mathcal{L}''(l) \end{cases}$$

Soit $L_0 = (L; (\Phi(k)(l) = a))$. Nous allons prouver que les hypothèses ci-dessus sont toujours vérifiées après la substitution de L par L_0. Cela terminera la preuve de ce cas, car cela entraîne $\Gamma \vdash \zeta(x : \alpha)[L_0 : \mathcal{L}]^{\varphi}_{\Phi} : \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}}$, et donc, par sous-typage, $\Gamma \vdash a' : \tau$.

Par inspection des règles de sous-typage, on voit que l'on doit avoir $\mathcal{A} = \emptyset$ et $\mathcal{K}' \subseteq \mathcal{K}$.

$$\begin{aligned} \operatorname{dom} \mathcal{L}' \setminus \operatorname{dom}(L_0 \circ \varphi) &= \operatorname{dom} \mathcal{L}' \setminus \operatorname{dom}((L; (\Phi(k)(l) = a)) \circ \varphi) \\ &\subseteq \operatorname{dom} \mathcal{L}' \setminus \operatorname{dom} L \circ \varphi \\ &\subseteq \mathcal{A} \subseteq \emptyset \end{aligned}$$

Par conséquent, $\operatorname{dom} \mathcal{L}' \setminus \operatorname{dom}(L_0 \circ \varphi) = \emptyset = \mathcal{A}$.

On a $l \in \operatorname{dom} \mathcal{L}'' = \operatorname{dom}(\mathcal{L} \circ \Phi(k))$, et donc, $\Phi(k)(l) \in \operatorname{dom} \mathcal{L}$. Alors :

$$\begin{aligned} \operatorname{dom} L_0 \cup \operatorname{img} \varphi &= \operatorname{dom}(L; (\Phi(k)(l) = a)) \cup \operatorname{img} \varphi \\ &= \operatorname{dom} L \cup \{\Phi(k)(l)\} \cup \operatorname{img} \varphi \\ &\supseteq \operatorname{dom} \mathcal{L} \cup \{\Phi(k)(l)\} \\ &\supseteq \operatorname{dom} \mathcal{L} \end{aligned}$$

Soit $l' \in \operatorname{dom} L_0$. Si l'on suppose que $l' \neq \Phi(k)(l)$, on a $L_0(l') = L(l')$. D'où, $\Gamma; (\alpha <\# \mathcal{K}); (x : \alpha) \vdash L_0(l') : \mathcal{L}(l')$. Supposons maintenant $l' = \Phi(k)(l)$. Alors $L_0(l') = a''$. D'autre part, $\Gamma; (\alpha <\# \mathcal{K}''); (x : \alpha) \vdash a'' : \mathcal{L}''(l)$ et $\mathcal{L}''(l) = (\mathcal{L} \circ \Phi(k))(l) = \mathcal{L}(l')$. D'après la règle Filtrage-Obj, $\mathcal{K}'' \subseteq \mathcal{K}'$. Donc, $\mathcal{K}'' \subseteq \mathcal{K}$. D'où, de nouveau, par le lemme 20, $\Gamma; (\alpha <\# \mathcal{K}); (x : \alpha) \vdash L_0(l') : \mathcal{L}(l')$.

Les autres prémisses sont inchangées.

- $v.l \longrightarrow L(\varphi(l))\{\tau''/\alpha\}\{v/x\}$ où $v = \zeta(x : \alpha)[L : \mathcal{L}]^{\varphi}_{\Phi}$ et $\tau'' = \zeta(\alpha)[\mathcal{L} \circ \varphi]_{\operatorname{dom}\Phi}$
La preuve de la première hypothèse doit se terminer par l'usage de la règle Sélection. Par inspection, cette règle est précédée de sous-typage et de la règle Proto. On a donc
$$\begin{cases} \tau = (\mathcal{L}_1(l))\{\tau'/\alpha\} \\ \tau' = \zeta(\alpha)[\mathcal{L}_1]_{\mathcal{K}_1} \\ \Gamma \vdash \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}} \preceq \tau' \\ \mathcal{L}' = \mathcal{L} \circ \varphi \\ \mathcal{A} = \operatorname{dom}\mathcal{L}' \setminus \operatorname{dom}(L \circ \varphi) \\ \mathcal{K} = \operatorname{dom}\Phi \\ \operatorname{dom}\mathcal{L} \subseteq \operatorname{dom} L \cup \operatorname{img}\varphi \\ \text{Quel que soit } l' \text{ dans } \operatorname{dom} L,\ \Gamma; (\alpha <\# \mathcal{K}); (x : \alpha) \vdash L(l') : \mathcal{L}(l') \\ \forall k \in \operatorname{dom}\Phi \cdot (k : \zeta(\alpha)[\mathcal{L}'']) \in \Gamma \wedge \mathcal{L}'' = \mathcal{L} \circ \Phi(k) \end{cases}$$
Par inspection des règles de sous-typage, on voit que $\mathcal{L}_1 \subseteq \mathcal{L}'$. Donc, $\mathcal{L}_1(l) = \mathcal{L}'(l) = \mathcal{L}(\varphi(l))$. On a donc $\Gamma; (\alpha <\# \mathcal{K}); (x : \alpha) \vdash L(\varphi(l)) : \mathcal{L}_1(l)$. De plus, $\Gamma \vdash \tau'' <\# \mathcal{K}$. Donc, en utilisant le lemme 21, $\Gamma; (x : \tau'') \vdash L(\varphi(l))\{\tau''/\alpha\} : \mathcal{L}_1(l)\{\tau''/\alpha\}$.
D'autre part, on voit que $\tau'' = \zeta(\alpha)[\mathcal{L}']_{\mathcal{K}}$, et donc, $\Gamma \vdash v : \tau''$. Par le lemme 23, $\Gamma \vdash L(\varphi(l))\{\tau''/\alpha\}\{v/x\} : \mathcal{L}_1(l)\{\tau''/\alpha\}$.
On a $\Gamma \vdash \tau'' \preceq \tau'$. Supposons d'abord que α apparaît en position covariante dans $\mathcal{L}_1$. Alors, par le lemme 26, $\Gamma \vdash L(\varphi(l))\{\tau''/\alpha\}\{v/x\} : \mathcal{L}_1(l)\{\tau'/\alpha\}$ comme désiré. Autrement, par inspection des règles de sous-typage, on voit que l'on a $\tau'' = \tau'$ (aucune règle ne s'applique). D'où le résultat.
- $v._k l \longrightarrow L(\Phi(k)(l))\{\tau''/\alpha\}\{v/x\}$ où $v = \zeta(x : \alpha)[L : \mathcal{L}]^{\varphi}_{\Phi}$ et $\tau'' = \zeta(\alpha)[\mathcal{L} \circ \varphi]_{\operatorname{dom}\Phi}$
La preuve de la première hypothèse doit se terminer par l'usage de la règle Vue-Sélection. Par inspection, cette règle est précédée par du sous-typage et la règle Proto. On a donc
$$\begin{cases} \tau = (\mathcal{L}''(l))\{\tau'/\alpha\} \\ \tau' = \zeta(\alpha)[\mathcal{L}_1]_{\mathcal{K}'} \\ \Gamma \vdash \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}} \preceq \tau' \\ \mathcal{L}' = \mathcal{L} \circ \varphi \\ \mathcal{A} = \operatorname{dom}\mathcal{L}' \setminus \operatorname{dom}(L \circ \varphi) \\ \mathcal{K} = \operatorname{dom}\Phi \\ \operatorname{dom}\mathcal{L} \subseteq \operatorname{dom} L \cup \operatorname{img}\varphi \\ \text{Quel que soit } l' \text{ dans } \operatorname{dom} L',\ \Gamma; (\alpha <\# \mathcal{K}); (x : \alpha) \vdash L(l') : \mathcal{L}(l') \\ \forall k \in \operatorname{dom}\Phi \cdot (k : \zeta(\alpha)[\mathcal{L}'']) \in \Gamma \wedge \mathcal{L}'' = \mathcal{L} \circ \Phi(k) \\ \Gamma \vdash \tau' <\# \mathcal{K}'' \\ k \in \mathcal{K}'' \\ (k : \zeta(\alpha)[\mathcal{L}'']) \in \Gamma \end{cases}$$
Par inspection des règles de sous-typage, on voit que $\mathcal{K}' \subseteq \mathcal{K}$. D'autre part, d'après la règle Filtrage-Obj, $\mathcal{K}'' \subseteq \mathcal{K}'$. D'où, $k \in \mathcal{K}$ et donc $\mathcal{L}''(l) = \mathcal{L}(\Phi(k)(l))$. Par conséquent, $\Gamma; (\alpha <\# \mathcal{K}); (x : \alpha) \vdash L(\Phi(k)(l)) : \mathcal{L}''(l)$. De plus, $\Gamma \vdash \tau'' <\# \mathcal{K}$. Donc, par le lemme 21, $\Gamma; (x : \tau'') \vdash L(\Phi(k)(l))\{\tau''/\alpha\} : \mathcal{L}''(l)\{\tau''/\alpha\}$.

D'autre part, on voit que $\tau'' = \zeta(\alpha)[\mathcal{L}']_{\mathcal{K}}$, et donc, $\Gamma \vdash v : \tau''$. Par le lemme 23, $\Gamma \vdash L(\Phi(k)(l))\{\tau''/\alpha\}\{v/x\} : \mathcal{L}''(l)\{\tau''/\alpha\}$.

On a $\Gamma \vdash \tau'' \preceq \tau'$. Supposons d'abord que α apparaît en position covariante dans $\mathcal{L}''$. Alors, en utilisant le lemme 26, $\Gamma \vdash L(\Phi(k)(l))\{\tau''/\alpha\}\{v/x\} : \mathcal{L}''(l)\{\tau'/\alpha\}$ comme désiré. Autrement, par inspection des règles de sous-typage, on voit que l'on a $\tau'' = \tau'$ (aucune règle ne s'applique). D'où le résultat.

– $\langle\zeta(x:\alpha)[L:\mathcal{L}]^{\varphi}_{\Phi}\rangle_k \longrightarrow \zeta(x:\alpha)[L:\mathcal{L}]^{\varphi}_{\Phi;(k=\varphi)}$

La preuve de la première hypothèse doit se terminer par l'usage de la règle Vue-Capture. On a donc $(k : \zeta(\alpha)[\mathcal{L}_1]) \in \Gamma$ et $\Gamma \vdash \zeta(x:\alpha)[L:\mathcal{L}]^{\varphi}_{\Phi} : \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}}$ pour des $\mathcal{L}'$, $\mathcal{A}$ et $\mathcal{K}$ tels que $\mathcal{L}_1 \subseteq \mathcal{L}'$ et $\tau = \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}\cup\{k\}}$. Le dernier jugement peut être montré à l'aide de la règle Proto. D'où :

$$\begin{cases} \mathcal{L}' = \mathcal{L} \circ \varphi \\ \mathcal{A} = \operatorname{dom}\mathcal{L}' \setminus \operatorname{dom}(L \circ \varphi) \\ \mathcal{K} = \operatorname{dom}\Phi \\ \operatorname{dom}\mathcal{L} \subseteq \operatorname{dom} L \cup \operatorname{img}\varphi \\ \text{Quel que soit } l \text{ dans } \operatorname{dom} L',\ \Gamma;(\alpha <\# \mathcal{K});(x:\alpha) \vdash L(l) : \mathcal{L}(l) \\ \forall k \in \operatorname{dom}\Phi \cdot (k : \zeta(\alpha)[\mathcal{L}'']) \in \Gamma \wedge \mathcal{L}'' = \mathcal{L} \circ \Phi(k) \end{cases}$$

Soient $\Phi_0 = \Phi; (k = \varphi)$ et $\mathcal{K}_0 = \mathcal{K} \cup \{k\}$. Nous montrer prouver que les hypothèses ci-dessus sont toujours vérifiées après la substitution de Φ par Φ_0 et de $\mathcal{K}$ par $\mathcal{K}$. Cela terminera la preuve de ce cas, car cela entraîne $\Gamma \vdash \zeta(x:\alpha)[L:\mathcal{L}]^{\varphi}_{\Phi_0} : \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}_0}$, c'est-à-dire, $\Gamma \vdash a' : \tau$.

On a clairement $\mathcal{K}_0 = \operatorname{dom}\Phi_0$.

De plus, $\mathcal{K} \subseteq \mathcal{K}_0$. Donc, d'après le lemme 20, pour tout l dans $\operatorname{dom} L$, $\Gamma;(\alpha <\# \mathcal{K}_0);(x:\alpha) \vdash L(l) : \mathcal{L}(l)$

Finalement, $(k : \zeta(\alpha)[\mathcal{L}_1]) \in \Gamma$ et $\mathcal{L}_1 \subseteq \mathcal{L}' = \mathcal{L} \circ \varphi = \mathcal{L} \circ \Phi_0(k)$.

Les autres prémisses sont inchangées.

– $\zeta(x:\alpha)[L:\mathcal{L}]^{\varphi}_{\Phi}|_k \longrightarrow \zeta(x:\alpha)[L:\mathcal{L}]^{\Phi(k)}_{\Phi}$

La preuve de la première hypothèse doit se terminer par l'usage de la règle Vue-Remplace. Par inspection, cette règle est précédée de sous-typage et de la règle Proto. On a donc

$$\begin{cases} \tau = \zeta(\alpha)[\mathcal{L}'']_{\mathcal{K}''} \\ \Gamma \vdash \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}} \preceq \zeta(\alpha)[\mathcal{L}_1]_{\mathcal{K}''} \\ k \in \mathcal{K}'' \\ (k : \zeta(\alpha)[\mathcal{L}'']) \in \Gamma \\ \mathcal{L}' = \mathcal{L} \circ \varphi \\ \mathcal{A} = \operatorname{dom}\mathcal{L}' \setminus \operatorname{dom}(L \circ \varphi) \\ \mathcal{K} = \operatorname{dom}\Phi \\ \operatorname{dom}\mathcal{L} \subseteq \operatorname{dom} L \cup \operatorname{img}\varphi \\ \text{Quel que soit } l' \text{ dans } \operatorname{dom} L',\ \Gamma;(\alpha <\# \mathcal{K});(x:\alpha) \vdash L(l') : \mathcal{L}(l') \\ \forall k \in \operatorname{dom}\Phi \cdot (k : \zeta(\alpha)[\mathcal{L}'']) \in \Gamma \wedge \mathcal{L}'' = \mathcal{L} \circ \Phi(k) \end{cases}$$

Soient $\mathcal{L}'_0 = \mathcal{L} \circ \Phi(k)$ et $\varphi_0 = \Phi(k)$. Nous allons prouver que les hypothèses ci-dessus sont toujours vérifiées après la substitution de $\mathcal{L}'$ par $\mathcal{L}$ et de φ par φ_0. Ainsi nous aurons $\Gamma \vdash \zeta(x:\alpha)[L:\mathcal{L}]^{\varphi_0}_{\Phi} : \zeta(\alpha)[\mathcal{L}'_0]^{\mathcal{A}}_{\mathcal{K}}$ et nous pourrons conclure par sous-typage, pourvu que $\Gamma \vdash \zeta(\alpha)[\mathcal{L}'_0]^{\mathcal{A}}_{\mathcal{K}} \preceq \zeta(\alpha)[\mathcal{L}'']_{\mathcal{K}''}$.

Clairement, $\mathcal{L}'_0 = \mathcal{L} \circ \varphi_0$.
On déduit de la seconde hypothèse que $\mathcal{A} = \emptyset$. D'où, $\mathrm{dom}(\mathcal{L} \circ \varphi) = \mathrm{dom}\,\mathcal{L}' \subseteq \mathrm{dom}(L \circ \varphi)$. Et donc, $\mathrm{dom}\,\mathcal{L} \cap \mathrm{img}\,\varphi \subseteq \mathrm{dom}\,L \cap \mathrm{img}\,\varphi \subseteq \mathrm{dom}\,L$. Alors, à l'aide de l'hypothèse, $\mathrm{dom}\,\mathcal{L} \subseteq \mathrm{dom}\,L \cup \mathrm{img}\,\varphi$, on peut déduire que $\mathrm{dom}\,\mathcal{L} \subseteq \mathrm{dom}\,L$.
Donc :

$$\begin{array}{rcl} \mathrm{dom}\,\mathcal{L}'_0 \setminus \mathrm{dom}(L \circ \varphi_0) & = & \mathrm{dom}(\mathcal{L} \circ \Phi(k)) \setminus \mathrm{dom}(L \circ \Phi(k)) \\ & = & \emptyset \\ & = & \mathcal{A} \end{array}$$

Finalement,

$$\begin{array}{rcl} \mathrm{dom}\,\mathcal{L} & \subseteq & \mathrm{dom}\,L \\ & \subseteq & \mathrm{dom}\,L \cup \mathrm{img}\,\varphi_0 \end{array}$$

Les autres prémisses sont inchangées.
Nous allons maintenant montrer : $\Gamma \vdash \zeta(\alpha)[\mathcal{L}'_0]^{\mathcal{A}}_{\mathcal{K}} \preceq \zeta(\alpha)[\mathcal{L}'']_{\mathcal{K}''}$. Du jugement $\Gamma \vdash \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}} \preceq \zeta(\alpha)[\mathcal{L}_1]_{\mathcal{K}''}$, nous déduisons que $\mathcal{A} = \emptyset$ et $\mathcal{K}'' \subseteq \mathcal{K}$. À l'aide de la règle Sub-Proto, on est ramené à prouver : $\Gamma \vdash \zeta(\alpha)[\mathcal{L}'_0]_{\mathcal{K}} \preceq \zeta(\alpha)[\mathcal{L}'']_{\mathcal{K}''}$. Comme $k \in \mathcal{K}'' \subseteq \mathcal{K}$ et $(k : \zeta(\alpha)[\mathcal{L}'']) \in \Gamma$, on a $\mathcal{L}'' = \mathcal{L} \circ \Phi(k) = \mathcal{L}'_0$. Si $\mathcal{K} = \mathcal{K}''$, ce cas est fini. Sinon, le jugement $\Gamma \vdash \zeta(\alpha)[\mathcal{L}']_{\mathcal{K}} \preceq \zeta(\alpha)[\mathcal{L}_1]_{\mathcal{K}''}$ ne peut être prouver qu'a l'aide de la règle Sub-Objet. Donc, $\forall k \in \mathcal{K}'' \cdot (k : \zeta(\alpha)[\mathcal{L}'']) \in \Gamma \wedge \mathrm{co}_\alpha(\mathcal{L}'')$ En particulier, α n'apparaît en position covariante dans $\mathcal{L}''$. La règle Sub-Objet peut donc être utilisée pour montrer le jugement désiré : $\Gamma \vdash \zeta(\alpha)[\mathcal{L}'']_{\mathcal{K}} \preceq \zeta(\alpha)[\mathcal{L}'']_{\mathcal{K}''}$.

– $(\lambda(x : \tau)a)(v) \longrightarrow a\{v/x\}$
La preuve de la première hypothèse doit se terminer par l'usage de la règle App. Par inspection des règles de typage, on voit que la règle est précédée de sous-typage et de la règle Abs. On a donc

$$\left\{ \begin{array}{l} \Gamma \vdash v : \tau' \\ \Gamma \vdash \tau'_1 \to \tau_1 \preceq \tau' \to \tau \\ \Gamma; (x : \tau'_1) \vdash a : \tau_1 \end{array} \right.$$

Par inspection des règles de sous-typage, on voit que le second jugement peut être prouvé par l'une des règles Sub-Réfl ou Sub-Flèche. Dans les deux cas, on a $\Gamma \vdash \tau_1 \preceq \tau$ et $\Gamma \vdash \tau' \preceq \tau'_1$. Alors, par le règle Sub, il vient $\Gamma \vdash v : \tau'_1$. D'après le lemme 23, $\Gamma \vdash a\{v/x\} : \tau_1$. On conclut à l'aide de la règle Sub : $\Gamma \vdash a\{v/x\} : \tau$.

– $\texttt{open}\ (\texttt{pack}\ v\ \texttt{as}\ \exists(k)\tau\ \texttt{hiding}\ \mathcal{K})\ \texttt{as}\ [k, x]\ \texttt{in}\ a \longrightarrow a\{\mathcal{K}/k\}\{v/x\}$
La preuve de la première hypothèse doit se terminer par l'usage de la règle Open. On a donc $\Gamma \vdash \texttt{pack}\ v\ \texttt{as}\ \exists(k)\tau\ \texttt{hiding}\ \mathcal{K} : \exists(k)\tau'$ et $\Gamma; (k); (x : \tau') \vdash a'' : \tau$ Par inspection des règles de typage, on voit que le premier de ces jugements doit être prouvé à l'aide de la règle Pack. D'où, $\Gamma \vdash v : \tau'\{\mathcal{K}/k\}$.
D'après le lemme 22, $\Gamma; (x : \tau'\{\mathcal{K}/k\}) \vdash a''\{\mathcal{K}/k\} : \tau$. Alors, d'après le lemme 23, $\Gamma \vdash a''\{\mathcal{K}/k\}\{v/x\} : \tau$.

– $F[\rho(k:t)a] \longrightarrow \rho(k:t)F[a]$

On le montre par induction sur la taille du contexte. Nous n'allons considérer que le cas où $F[\_] = (\_)(a')$, les autres cas étant similaires.

Nous montrons donc $(\rho(k:t)a)(a') \longrightarrow \rho(k:t)(a(a'))$

On remarque tout d'abord que la règle VUE-ABS commute avec la règle SUB. Supposons en effet que nous ayons la dérivation suivante :

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma; (k:t) \vdash a : \tau}}{\Gamma \vdash \rho(k:t)a : \tau}\ (\textsc{Vue-Abs}) \qquad \dfrac{\vdots}{\Gamma \vdash \tau \preceq \tau'}}{\Gamma \vdash \rho(k:t)a : \tau'}\ (\textsc{Sub})$$

D'après le lemme 19, $\Gamma; (k:t) \vdash \tau \preceq \tau'$. Donc :

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma; (k:t) \vdash a : \tau} \qquad \dfrac{\vdots}{\Gamma; (k:t) \vdash \tau \preceq \tau'}}{\Gamma; (k:t) \vdash a : \tau'}\ (\textsc{Sub})}{\Gamma \vdash \rho(k:t)a : \tau'}\ (\textsc{Vue-Abs})$$

On peut donc supposer que la règle VUE-ABS n'est pas suivie de sous-typage.

La preuve de la première hypothèse doit se terminer par l'utilisation de la règle EXTENSION, précédée de la règle VUE-ABS :

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma; (k:t) \vdash a : \tau' \to \tau}}{\Gamma \vdash \rho(k:t)a : \tau' \to \tau}\ (\textsc{Vue-Abs}) \qquad \dfrac{\vdots}{\Gamma \vdash a' : \tau'}}{\Gamma \vdash \rho(k:t)a(a') : \tau}\ (\textsc{App})$$

D'après le lemme 19, $\Gamma; (k:t) \vdash a' : \tau'$. Donc :

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma; (k:t) \vdash a : \tau' \to \tau} \qquad \dfrac{\vdots}{\Gamma; (k:t) \vdash a' : \tau'}}{\Gamma; (k:t) \vdash a(a') : \tau}\ (\textsc{App})}{\Gamma \vdash \rho(k:t)a(a') : \tau}\ (\textsc{Vue-Abs})$$

■

Lemme 18 (Décomposition et remplacement) *Si* $\Gamma \vdash E[F[a]] : \tau$, *alors il existe un type* τ' *tel que* $\Gamma \vdash a : \tau'$ *et tel que de plus si* $\Gamma \vdash a' : \tau'$ *alors* $\Gamma \vdash E[F[a']] : \tau$.

Démonstration. Par induction sur une preuve de $\Gamma \vdash E[F[a]] : \tau$ ■

Lemme 19 (Renforcement de l'environnement) *Si* $\Gamma \vdash \mathcal{J}$ *et* $\Gamma \subseteq \Gamma'$, *alors* $\Gamma' \vdash \mathcal{J}$ *(où* $\mathcal{J}$ *représente la partie droite d'un jugement).*

Démonstration. Par induction sur une preuve de $\Gamma \vdash \mathcal{J}$ ■

Lemme 20 (Renforcement du filtrage) *If* $\Gamma; (\alpha <\# \mathcal{K}) \vdash a : \tau$ *et* $\mathcal{K} \subseteq \mathcal{K}'$ *alors* $\Gamma; (\alpha \preceq \mathcal{K}') \vdash a : \tau$.

Démonstration. Par induction sur une preuve de la première prémisse.

Une hypothèse $\alpha <\# \mathcal{K}$ ne peut être utilisée que par la règle FILTRAGE-VAR :

$$\frac{(\alpha <\# \mathcal{K}) \in \Gamma \qquad \mathcal{K}_1 \subseteq \mathcal{K}}{\Gamma \vdash \alpha <\# \mathcal{K}_1} \ (\text{Filtrage-Var})$$

Par transitivité, $\mathcal{K}_1 \subseteq \mathcal{K}'$. Donc :

$$\frac{(\alpha <\# \mathcal{K}) \in \Gamma \qquad \mathcal{K}_1 \subseteq \mathcal{K}'}{\Gamma \vdash \alpha <\# \tau_1} \ (\text{Filtrage-Var})$$

Les autres cas de la preuves sont évidents. ■

Lemme 21 (Élimination du filtrage) *Si* $\Gamma; (\alpha <\# \mathcal{K}) \vdash a : \tau$ *et* $\Gamma' \vdash \tau' <\# \mathcal{K}$ *où* $\Gamma' \subseteq \Gamma$ *alors* $\Gamma\{\tau'/\alpha\} \vdash a\{\tau'/\alpha\} : \tau\{\tau'/\alpha\}$.

Démonstration. Par induction sur une preuve de la première prémisse. On montre simultanément que si $\Gamma; (\alpha <\# \mathcal{K}) \vdash \tau_1 \preceq \tau_2$ et $\Gamma' \vdash \tau' \preceq \mathcal{K}$ où $\Gamma' \subseteq \Gamma$ alors $\Gamma\{\tau'/\alpha\} \vdash \tau_1\{\tau'/\alpha\} \preceq \tau_2\{\tau'/\alpha\}$.

Le cas le plus difficile est celui de la règle FILTRAGE-VAR.

$$\frac{(\alpha_1 <\# \mathcal{K}_1) \in \Gamma; (\alpha <\# \mathcal{K}) \qquad \mathcal{K}' \subseteq \mathcal{K}_1}{\Gamma; (\alpha <\# \mathcal{K}) \vdash \alpha_1 <\# \mathcal{K}'} \ (\text{Filtrage-Var})$$

Si $\alpha_1 \neq \alpha$, on a $(\alpha_1 <\# \mathcal{K}_1) \in \Gamma\{\tau'/\alpha\}$. D'où, $\Gamma\{\tau'/\alpha\} \vdash \alpha_1\{\tau'/\alpha\} <\# \mathcal{K}'$ comme désiré.
Supposons maintenant que $\alpha_1 = \alpha$. Alors, $\mathcal{K}_1 = \mathcal{K}$. Comme α n'apparaît pas libre dans la seconde prémisse, $\Gamma'\{\tau'/\alpha\} \vdash \tau' <\# \mathcal{K}$. D'après le lemme 19, $\Gamma\{\tau'/\alpha\} \vdash \tau' <\# \mathcal{K}$. Par inspection des règles de filtrage, en utilisant le fait que $\mathcal{K}' \subseteq \mathcal{K}$, on conclut que $\Gamma\{\tau'/\alpha\} \vdash \tau' <\# \mathcal{K}'$ comme désiré.

Les autres cas sont faciles. ■

Lemme 22 (Élimination des vues abstraites) *Si* $\Gamma; (k) \vdash a : \tau$ *et* $\mathcal{K} \subseteq \operatorname{dom}\Gamma$ *alors* $\Gamma\{\mathcal{K}/k\} \vdash a\{\mathcal{K}/k\} : \tau\{\mathcal{K}/k\}$.

Démonstration. Par induction sur la preuve de la première hypothèse.

– Règle PROTO :

$$\frac{\begin{array}{ll} \mathcal{L}' = \mathcal{L} \circ \varphi & \mathcal{A} = \operatorname{dom}\mathcal{L}' \setminus \operatorname{dom}(L \circ \varphi) \\ \mathcal{K}' = \operatorname{dom}\Phi & \operatorname{dom}\mathcal{L} \subseteq \operatorname{dom} L \cup \operatorname{img}\varphi \\ \multicolumn{2}{l}{\text{Quel que soit } l \text{ dans } \operatorname{dom} L',\ \Gamma; (\alpha <\# \mathcal{K}); (x : \alpha) \vdash L(l) : \mathcal{L}(l)} \\ \multicolumn{2}{l}{\forall k' \in \operatorname{dom}\Phi \cdot (k' : \zeta(\alpha)[\mathcal{L}'']) \in \Gamma; (k) \wedge \mathcal{L}'' = \mathcal{L} \circ \Phi(k)} \end{array}}{\Gamma; (k) \vdash \zeta(x : \alpha)[L : \mathcal{L}]^{\varphi}_{\Phi} : \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}'}}$$

Pour toute vue $k' \in \mathcal{K}$, on a $(k' : \zeta(\alpha)[\mathcal{L}'']) \in \Gamma; (k)$. Par conséquent, $\mathcal{K}'\{\mathcal{K}'/k\} = \mathcal{K}'$. D'après l'hypothèse d'induction, on a pour toute méthode l de $\operatorname{dom} L$: $\Gamma\{\mathcal{K}/k\}; (\alpha <\# \mathcal{K}'); (x : \alpha) \vdash L(l) : \mathcal{L}(l)$. Les autres prémisses de cette règle sont clairement préservées par la substitution. L'application de la règle VUE-SÉLECTION nous permet donc de conclure dans ce cas.

– Règle Vue-Sélection :

$$\frac{\Gamma;(k)\vdash a:\tau \qquad \Gamma;(k)\vdash \tau <\# \mathcal{K}' \qquad k'\in\mathcal{K}' \qquad (k':\zeta(\alpha)[\mathcal{L}'])\in\Gamma;(k)}{\Gamma;(k)\vdash a._{k'}l:(\mathcal{L}'(l))\{\tau/\alpha\}}\ (\text{Vue-Sélection})$$

On a $(k':\zeta(\alpha)[\mathcal{L}'])\in\Gamma;(k)$. Par conséquent $k'\neq k$. Et donc, $k'\in\mathcal{K}'\{\mathcal{K}/k\}$. En utilisant l'hypothèse d'induction, il vient : $\Gamma\{\mathcal{K}/k\}\vdash\tau\{\mathcal{K}/k\}\preceq\zeta(\alpha)[\emptyset]_{\mathcal{K}'}\{\mathcal{K}/k\}$. Toutes les prémisses de cette règle sont clairement préservées par la substitution. L'application de la règle Vue-Sélection nous permet donc de conclure dans ce cas.

– Règles Vue-Redéfinition, Vue-Capture et Vue-Remplace : la preuve est semblable à celle du cas précédent.

– Règle Pack :

$$\frac{\Gamma;(k)\vdash a:\tau\{\mathcal{K}'/k'\}}{\Gamma;(k)\vdash \texttt{pack}\ a\ \texttt{as}\ \exists(k')\tau\ \texttt{hiding}\ \mathcal{K}':\exists(k')\tau}\ (\text{Pack})$$

Par renommage, on peut supposer que $k'\neq k$ et $k'\notin\mathcal{K}$. Par hypothèse d'induction, $\Gamma\{\mathcal{K}/k\}\vdash a\{\mathcal{K}/k\}:\tau\{\mathcal{K}'/k'\}\{\mathcal{K}/k\}$. D'après la remarque précédente, $\tau\{\mathcal{K}'/k'\}\{\mathcal{K}/k\}=\tau\{\mathcal{K}/k\}\{\mathcal{K}'\{\mathcal{K}/k\}/k'\}$. Par conséquent, en utilisant la règle Pack, $\Gamma\{\mathcal{K}/k\}\vdash\texttt{pack}\ a\{\mathcal{K}/k\}\ \texttt{as}\ \exists(k')(\tau\{\mathcal{K}/k\})\ \texttt{hiding}\ \mathcal{K}'\{\mathcal{K}/k\}:\exists(k')\tau\{\mathcal{K}/k\}$. C'est-à-dire,

$$\Gamma\{\mathcal{K}/k\}\vdash(\texttt{pack}\ a\ \texttt{as}\ \exists(k')\tau\ \texttt{hiding}\ \mathcal{K}')\{\mathcal{K}/k\}:(\exists(k')\tau)\{\mathcal{K}/k\}$$

comme désiré.

– Règle Sub-Objet

$$\frac{\mathcal{L}'\subseteq\mathcal{L} \qquad \mathcal{K}''\subseteq\mathcal{K}' \qquad \alpha \text{ apparaît en position covariante dans } \mathcal{L}'}{\Gamma;(k)\vdash\zeta(\alpha)[\mathcal{L}]_{\mathcal{K}'}\preceq\zeta(\alpha)[\mathcal{L}']_{\mathcal{K}''}}\ (\text{Sub-Objet})$$

On a $\mathcal{K}''\{\mathcal{K}/k\}\subseteq\mathcal{K}'\{\mathcal{K}/k\}$. Par ailleurs, les autres prémisses de cette règle sont clairement préservées par la substitution. L'application de la règle Vue-Sélection nous permet donc de conclure dans ce cas.

– Les autres cas sont faciles.

■

Lemme 23 (Substitution de valeur) *Si* $\Gamma;(x:\tau)\vdash a':\tau'$ *et* $\Gamma\vdash a:\tau$, *alors* $\Gamma\vdash a'\{a/x\}:\tau'$.

Démonstration. Par induction sur la preuve de la première hypothèse.
La preuve est standard et ne présente pas de difficulté particulière. ■

Lemme 24 (Transitivité du sous-typage) *Si* $\Gamma\vdash\tau\preceq\tau'$ *et* $\Gamma\vdash\tau'\preceq\tau''$ *alors* $\Gamma\vdash\tau\preceq\tau''$.

Démonstration. Par induction sur des dérivations prouvant les hypothèses.

– Règle Sub-Réfl et toute règle : trivial.

- Règles SUB-FILTRAGE et SUB-OBJET : On a $\tau = \alpha$, $\tau' = \zeta(\alpha')[\emptyset]_{\mathcal{K}}$ et $\tau'' = \zeta(\alpha')[\emptyset]_{\mathcal{K}'}$ où $\mathcal{K}' \subseteq \mathcal{K}$. Par inspection des règles de filtrage, on voit que l'on peut directement en déduire que $\tau \preceq \tau''$.
- Règles SUB-FLÈCHE et SUB-FLÈCHE : Par application de l'hypothèse d'induction.
- Règles SUB-OBJET et SUB-OBJET : Par transitivité de l'inclusion.

■

Lemme 25 (Sous-typage des classes) *Si* $\Gamma \vdash \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}} \preceq \zeta(\alpha)[\mathcal{L}']_{\mathcal{K}'}$*, alors* $\mathcal{A} = \emptyset$*,* $\mathcal{L}' \subseteq \mathcal{L}$ *et* $\mathcal{K}' \subseteq \mathcal{K}$*.*

Démonstration. Par induction sur une preuve de l'hypothèse. ■

Lemme 26 (Substitution en position covariante) *Si α apparaît en position covariante dans τ et si* $\Gamma \vdash \tau' \preceq \tau''$*, alors* $\Gamma \vdash \tau\{\tau'/\alpha\} \preceq \tau\{\tau''/\alpha\}$*.*

Démonstration. Par induction sur une preuve de la seconde hypothèse. On prouve simultanément le jugement ci-dessus et le jugement suivant :

> *Si α apparaît en position contravariante dans τ et si* $\Gamma \vdash \tau' \preceq \tau''$*,*
> *alors* $\Gamma \vdash \tau\{\tau''/\alpha\} \preceq \tau\{\tau'/\alpha\}$*.*

Considérons d'abord le premier jugement. Si α n'est pas libre dans τ, la conclusion se déduit de SUB-RÉFL. Si $\tau = \alpha$, alors la conclusion est donnée par la seconde hypothèse $\Gamma \vdash \tau' \preceq \tau''$. Autrement, $\tau = \tau_1 \to \tau_2$ où α apparaît en position contravariante dans τ_1 et en position covariante dans τ_2. D'après l'hypothèse d'induction, $\Gamma \vdash \tau_1\{\tau''/\alpha\} \preceq \tau_1\{\tau'/\alpha\}$ et $\Gamma \vdash \tau_2\{\tau'/\alpha\} \preceq \tau_2\{\tau''/\alpha\}$. La règle SUB-FLÈCHE permet de conclure.

Considérons maintenant le second jugement. Si α n'est pas libre dans τ, alors la règle SUB-RÉFL permet de conclure. Autrement, $\tau = \tau_1 \to \tau_2$ où α apparaît en position covariante dans τ_1 et contravariante dans τ_2. D'après l'hypothèse d'induction, $\Gamma \vdash \tau_1\{\tau'/\alpha\} \preceq \tau_1\{\tau''/\alpha\}$ et $\Gamma \vdash \tau_2\{\tau''/\alpha\} \preceq \tau_2\{\tau'/\alpha\}$. La règle SUB-FLÈCHE permet de conclure. ■

Progression

Théorème 27 (Progression) *Si* $\vdash a : \tau$ *alors a est de la forme $E[v]$ pour une certaine valeur v, ou alors* $a \longrightarrow a'$ *pour une certaine expression a'.*

La preuve de ce résultat, s'appuie sur le lemme suivant :

Lemme 28 (Type des valeurs) *Le type des valeurs permet de les classer :*

- *si* $\Gamma \vdash v : \tau' \to \tau$ *alors v est une fonction* $\lambda(x : \tau)a$ *;*
- *si* $\Gamma \vdash v : t_{\mathcal{K}}$ *ou* $\Gamma \vdash v : t^{\mathcal{A}}_{\mathcal{K}}$ *alors v est un objet* $\zeta(x : \alpha)[L : \mathcal{L}]^{\varphi}_{\Phi}$ *;*
- *si* $\Gamma \vdash v : \exists(k)\tau$ *alors v est une abstraction de vue* `pack` a `as` τ `hiding` $\mathcal{K}$*.*

Démonstration. Par inspection des règles de typage, on voit que

- Si v est une fonction $\lambda(x : a)$, alors son type est de la forme $\tau' \to \tau$;
- Si v est un objet $\zeta(x : \alpha)[L : \mathcal{L}]^{\varphi}_{\Phi}$, alors son type est de la forme $t_{\mathcal{K}}$ ou $t^{\mathcal{A}}_{\mathcal{K}}$;
- Si v est une abstraction de vue `pack` a `as` τ `hiding` $\mathcal{K}$ alors son type est de la forme $\exists(k)\tau$.

Comme on a ainsi considéré tous les cas de valeurs et que les formes des types sont disjoints, on en déduit la forme d'une valeur en fonction de son type. ∎

Nous pouvons maintenant montrer l'absence de bloquage :
Démonstration. Supposons que $\vdash a : \tau$ et a ne soit pas de la forme $E[v]$. Considérons un contexte d'évaluation F tel que $a = E[F[a_0]]$ et a_0 n'est pas une valeur. D'après le lemme 18, $\vdash a_0 : \tau'$ pour un certain type τ'. Si le lemme est vrai pour a_0 alors $a_0 \longrightarrow a_0'$ et donc $a \longrightarrow E[a_0']$. On peut donc supposer que le plus grand contexte d'évaluation F tel que $a = E[F[a_0]]$ et a_0 n'est pas une valeur soit le contexte vide.

On conclut en considérant les formes possibles de a. Il n'y a pas de difficulté particulière. Nous ne présentons donc qu'un cas typique. Les autres cas se montre de façon similaire.

- $a = a_1.l \Leftarrow \varsigma(x : \alpha)a_1'$

  Nécessairement, a_1 est une valeur. De plus, une dérivation de $\vdash a : \tau$ contient la règle suivante :

  $$\frac{\begin{array}{c}\Gamma \vdash a_1 : \zeta(\alpha)[\mathcal{L}]_{\mathcal{K}}^{\mathcal{A}} \\ \Gamma; (\alpha <\# \mathcal{K}); (x : \alpha) \vdash a_1' : \mathcal{L}(l)\end{array}}{\Gamma \vdash a_1.l \Leftarrow \varsigma(x : \alpha)a_1' : \zeta(\alpha)[\mathcal{L}]_{\mathcal{K}}^{\mathcal{A}\setminus\{l\}}} \ (\textsc{Redéfinition})$$

  La valeur a_1 a un type d'objet. C'est donc un objet $\zeta(x : \alpha)[L : \mathcal{L}_1]_{\Phi}^{\varphi}$. De plus, par inspection des règles de sous-typage et de la règle Proto, on voit que l'on a $\operatorname{dom} L \subseteq \operatorname{dom} \varphi$. Par conséquent, $l \in \operatorname{dom} \varphi$, et la règle de réduction définissant la redéfinition d'une méthode peut être appliquée, donnant lieu à l'expression $a' = \zeta(x : \alpha)[L; (\varphi(l) = a_1') : \mathcal{L}_1]_{\Phi}^{\varphi}$

∎

5.3 Langage de classe

5.3.1 Syntaxe

Cette section présente une extension du calcul de base avec des classes. La syntaxe des classes, présentée dans la figure 5.6 est classique. Cependant, l'ajout d'une méthode est distinct de sa définition. Ce choix a été fait de manière à rendre la traduction du langage dans le calcul de base plus simple.

Afin d'être plus compréhensible, les exemples au début de ce chapitre faisaient usage de sucre syntaxique : l'addition d'une méthode et sa définition sont combinés en une construction unique ; la variable représentant une instance de la classe, et la variable de type représentant son type, ne sont liés qu'une fois plutôt qu'à chaque définition de méthode.

5.3.2 Traduction

Définition

Nous définissons maintenant une traduction typée du langage dans le calcul de base. Pour cela, les environnements sont étendus avec des liaisons de classes :

$a ::=$	$\ldots$	
	$\mid \texttt{class}\ c\ [k](x:\tau) = \mathit{expr}\ \texttt{in}\ a$	Classe privée
	$\mid \texttt{class}\ c\ [\texttt{protected}\ k](x:\tau) = \mathit{expr}\ \texttt{in}\ a$	Classe protégée
	$\mid \texttt{class}\ c\ [\texttt{public}\ k](x:\tau) = \mathit{expr}\ \texttt{in}\ a$	Classe publique
	$\mid \texttt{new}\ c$	Constructeur d'objets
$\mathit{expr} ::=$	$(\mathit{expr} : \mathit{type})$	Contrainte de classe
	$\mid \texttt{object}\ \mathit{corps}\ \texttt{end}$	Corps de classe
$\mathit{corps} ::=$	$\mathit{extension}$	
	$\mid \mathit{corps}\ \texttt{method}\ l = \varsigma(x:\alpha)a$	Définition de méthode
$\mathit{extension} ::=$	$\textit{héritage}$	
	$\mid \mathit{extension}\ \texttt{abstract}\ l : \varsigma(\alpha)\tau$	Ajout d'une méthode
$\textit{héritage} ::=$	$\emptyset$	
	$\mid \texttt{inherit}\ c(a)$	Héritage
$\mathit{type} ::=$	$\texttt{object}\ \textit{type-corps}\ \texttt{end}$	Type de classe
$\textit{type-corps} ::=$	$\textit{type-hérit}$	
	$\mid \textit{type-corps}\ \texttt{abstract}\ l : \varsigma(\alpha)\tau$	Méthode abstraite
	$\mid \textit{type-corps}\ \texttt{method}\ l : \varsigma(\alpha)\tau$	Méthode concrète
$\textit{type-hérit} ::=$	$\emptyset$	
	$\mid \texttt{inherit}\ c$	Héritage (types)

FIG. 5.6: Syntaxe des classes

(TYPE-VIDE)
$$\frac{}{\Gamma \vdash \emptyset : \zeta(\alpha)[\emptyset]^{\emptyset}_{\emptyset}}$$

(TYPE-HÉRITAGE)
$$\frac{(c : (\mathcal{K}', \tau' \to \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}})) \in \Gamma}{\Gamma \vdash \texttt{inherit}\ c : \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}\setminus\mathcal{K}'}}$$

(TYPE-MÉTHODE-ABSTRAITE)
$$\frac{\Gamma \vdash \mathit{corps} : \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}}{\Gamma \vdash (\mathit{corps}\ \texttt{abstract}\ l : \varsigma(\alpha)\tau) : \zeta(\alpha)[\mathcal{L}; (l : \tau)]^{\mathcal{A}\cup\{l\}}_{\mathcal{K}}}$$

(TYPE-MÉTHODE-CONCRÈTE)
$$\frac{\Gamma \vdash \mathit{corps} : \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}}{\Gamma \vdash (\mathit{corps}\ \texttt{method}\ l : \varsigma(\alpha)\tau) : \zeta(\alpha)[\mathcal{L}; (l : \tau)]^{\mathcal{A}\setminus\{l\}}_{\mathcal{K}}}$$

(TYPE-CLASSE)
$$\frac{\Gamma \vdash \mathit{type\text{-}corps} : \tau}{\Gamma \vdash \texttt{object}\ \mathit{type\text{-}corps}\ \texttt{end} : \tau}$$

FIG. 5.7: Traduction des types de classes

$$\begin{array}{ll} \Gamma ::= \ldots & \\ \quad \mid \Gamma; (c : (\mathcal{K}', \tau \to \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}})) & \text{Liaison de classe} \end{array}$$

Le type d'une classe est une paire $(\mathcal{K}', \tau \to \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}})$. Le type τ est le type de l'argument de la classe, tandis que $\zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}$ représente le type du corps de la classe. L'ensemble $\mathcal{K}'$ contient les vues de la classe qui devront être masquées. En effet, la sémantique statique du calcul de base ne permet pas de masquer les vues d'un prototype : il est seulement possible de regrouper un ensemble de vues en une vue abstraite. L'ensemble $\mathcal{K}'$ sert donc à garder trace des vues qui devront être cachées dans le type des objets de la classe (qui est donc $\zeta(\alpha)[\mathcal{L}]_{\mathcal{K}\setminus\mathcal{K}'}$).

Les règles de traduction sont données dans les figures 5.7, 5.8 et 5.9. Elles définissent les jugements suivants :

- $\Gamma \vdash \mathit{type} : \tau$, $\Gamma \vdash \mathit{type\text{-}corps} : \tau$ et $\Gamma \vdash \mathit{type\text{-}hérit} : \tau$ pour le type des classes ;
- $\Gamma \vdash \mathit{expr} : (\mathcal{K}, \tau) \Rightarrow \overline{a}$, $\Gamma \vdash \mathit{corps} : (\mathcal{K}, \tau) \Rightarrow \overline{a}$, $\Gamma \vdash \mathit{extension} : (\mathcal{K}, \tau) \Rightarrow \overline{a}$ et $\Gamma \vdash \mathit{héritage} : (\mathcal{K}, \tau) \Rightarrow \overline{a}$ pour les expressions de classe ;
- $\Gamma \vdash a : \tau \Rightarrow \overline{a}$ pour les autres expressions.

Nous avons omis les règles de traduction des expressions du calcul de base car elles sont claires. Ainsi, voici par exemple celle correspondant à la redéfinition d'une méthode :

$$\frac{\Gamma \vdash a : \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}} \Rightarrow \overline{a} \qquad \Gamma; (\alpha <\#\ \mathcal{K}); (x : \alpha) \vdash a' : \mathcal{L}(l) \Rightarrow \overline{a'}}{\Gamma \vdash a.l \Leftarrow \varsigma(x : \alpha)a' : \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}\setminus\{l\}}_{\mathcal{K}} \Rightarrow \overline{a}.l \Leftarrow \varsigma(x : \alpha)\overline{a'}}$$

(Classe-Vide)

$$\frac{}{\Gamma \vdash \emptyset : (\emptyset, \zeta(\alpha)[\emptyset]^{\emptyset}_{\emptyset}) \Rightarrow \zeta(x : \alpha)[\emptyset : \emptyset]^{\emptyset}_{\emptyset}}$$

(Héritage)

$$\frac{(c : (\mathcal{K}, \tau' \to \tau)) \in \Gamma \qquad \Gamma \vdash a : \tau'_1 \Rightarrow \overline{a} \qquad \Gamma \vdash \tau'_1 \preceq \tau'}{\Gamma \vdash \texttt{inherit}\ c(a) : (\mathcal{K}, \tau) \Rightarrow c(\overline{a})}$$

(Méthode-Abstraite)

$$\frac{\Gamma \vdash \mathit{extension} : (\mathcal{K}', \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}) \Rightarrow a}{\Gamma \vdash (\mathit{extension}\ \texttt{abstract}\ l : \varsigma(\alpha)\tau) : (\mathcal{K}', \zeta(\alpha)[\mathcal{L}; (l : \tau)]^{\mathcal{A} \cup \{l\}}_{\mathcal{K}}) \Rightarrow a + (l : \varsigma(\alpha)\tau)}$$

(Vue-Locale)

$$\frac{\Gamma \vdash \mathit{extension} : (\mathcal{K}', \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}) \Rightarrow a}{\Gamma, (k : \zeta(\alpha)[\mathcal{L}]) \vdash \mathit{extension} : (\mathcal{K}' \cup \{k\}, \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K} \cup \{k\}}) \Rightarrow \langle a \rangle_k}$$

(Méthode)

$$\frac{\Gamma, (k : \zeta(\alpha)[\mathcal{L}]) \vdash \mathit{corps} : (\mathcal{K}', \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}) \Rightarrow a' \qquad \Gamma; (k : \zeta(\alpha)[\mathcal{L}]); (\alpha <\# \mathcal{K}); (x : \alpha) \vdash a : \mathcal{L}(l) \Rightarrow \overline{a}}{\Gamma, (k : \zeta(\alpha)[\mathcal{L}]) \vdash \mathit{corps}\ \texttt{method}\ l = \varsigma(x : \alpha)a : (\mathcal{K}', \zeta(\alpha)[\mathcal{L}]^{\mathcal{A} \setminus \{l\}}_{\mathcal{K}}) \Rightarrow a'.l \Leftarrow \varsigma(x : \alpha)\overline{a}}$$

(Classe-Corps)

$$\frac{\Gamma, (k : \zeta(\alpha)[\mathcal{L}]) \vdash \mathit{corps} : (\mathcal{K}, \tau) \Rightarrow a}{\Gamma, (k : \zeta(\alpha)[\mathcal{L}]) \vdash \texttt{object}\ \mathit{corps}\ \texttt{end} : (\mathcal{K}, \tau)\tau \Rightarrow a}$$

(Classe-Contrainte)

$$\frac{\Gamma, (k : \zeta(\alpha)[\mathcal{L}]) \vdash \mathit{expr} : (\mathcal{K}', \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}) \Rightarrow a \qquad \Gamma \vdash \mathit{type} : \zeta(\alpha)[\mathcal{L}_1]^{\mathcal{A}}_{\mathcal{K}_1} \quad \mathcal{L}_1 \subseteq \mathcal{L} \quad \mathcal{K}_1 \subseteq \mathcal{K} \quad \mathcal{A} \subseteq \operatorname{dom}\mathcal{L}_1}{\Gamma, (k : \zeta(\alpha)[\mathcal{L}]) \vdash (\mathit{expr} : \mathit{type}) : (\mathcal{K} \setminus \mathcal{K}_1, \zeta(\alpha)[\mathcal{L}_1]^{\mathcal{A}}_{\mathcal{K}}) \Rightarrow a \setminus (\operatorname{dom}\mathcal{L} \setminus \operatorname{dom}\mathcal{L}_1)}$$

Fig. 5.8: Traduction des classes

(Classe-Privée)

$$\frac{\begin{array}{c}\Gamma;(x:\tau'),(k:\zeta(\alpha)[\mathcal{L}]) \vdash \mathit{expr} : (\mathcal{K}',\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}}) \Rightarrow a' \\ \Gamma;(k);(c:(\{k\},\tau' \to \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}\setminus\mathcal{K}'\cup\{k\}})) \vdash a : \tau \Rightarrow \overline{a}\end{array}}{\begin{array}{c}\Gamma \vdash \texttt{class}\ c\ [k](x:\tau') = \mathit{expr}\ \texttt{in}\ a : \tau \Rightarrow \\ \texttt{open}\ (\rho(k:\zeta(\alpha)[\mathcal{L}])\texttt{pack}\ \lambda(x:\tau')a'\ \texttt{as}\ \exists(k')(\tau' \to \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}\setminus\mathcal{K}'\cup\{k'\}}) \\ \texttt{hiding}\ \mathcal{K}')\ \texttt{as}\ [k,c]\ \texttt{in}\ \overline{a}\end{array}}$$

(Classe-Protégée)

$$\frac{\begin{array}{c}\Gamma;(x:\tau'),(k:\zeta(\alpha)[\mathcal{L}]) \vdash \mathit{expr} : (\mathcal{K}',\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}}) \Rightarrow a' \\ \Gamma;(k);(c:(\emptyset,\tau' \to \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}\setminus\mathcal{K}'\cup\{k\}})) \vdash a : \tau \Rightarrow \overline{a}\end{array}}{\begin{array}{c}\Gamma \vdash \texttt{class}\ c\ [\texttt{protected}\ k](x:\tau') = \mathit{expr}\ \texttt{in}\ a : \tau \Rightarrow \\ \texttt{open}\ (\rho(k:\zeta(\alpha)[\mathcal{L}])\texttt{pack}\ \lambda(x:\tau')a'\ \texttt{as}\ \exists(k')(\tau' \to \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}\setminus\mathcal{K}'\cup\{k'\}}) \\ \texttt{hiding}\ \mathcal{K}')\ \texttt{as}\ [k,c]\ \texttt{in}\ \overline{a}\end{array}}$$

(Classe-Publique)

$$\frac{\begin{array}{c}\Gamma;(x:\tau'),(k:\zeta(\alpha)[\mathcal{L}]) \vdash \mathit{expr} : (\mathcal{K}',\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}}) \Rightarrow a' \\ \Gamma;(k'');(k':\zeta(\alpha)[\mathcal{L}']);(c:(\emptyset,\tau' \to \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}\setminus\mathcal{K}'\cup\{k',k''\}})) \vdash a\{k'/k\} : \tau \Rightarrow \overline{a}\end{array}}{\begin{array}{c}\Gamma \vdash \texttt{class}\ c\ [\texttt{public}\ k](x:\tau') = \mathit{expr}\ \texttt{in}\ a : \tau \Rightarrow \\ \rho(k':\zeta(\alpha)[\mathcal{L}'])\texttt{open}\ (\rho(k:\zeta(\alpha)[\mathcal{L}])\texttt{pack}\ \lambda(x:\tau')\langle a'\rangle_{k'} \\ \texttt{as}\ \exists(k_0)(\tau' \to \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}\setminus\mathcal{K}'\cup\{k_0,k'\}})\ \texttt{hiding}\ \mathcal{K}')\ \texttt{as}\ [k'',c]\ \texttt{in}\ \overline{a}\end{array}}$$

(New)

$$\frac{\begin{array}{c}(c:(\mathcal{K}',\tau' \to \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}})) \in \Gamma \\ \Gamma \vdash \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}} \preceq \zeta(\alpha)[\mathcal{L}]_{\mathcal{K}\setminus\mathcal{K}'}\end{array}}{\Gamma \vdash \texttt{new}\ c : \tau' \to \zeta(\alpha)[\mathcal{L}]_{\mathcal{K}\setminus\mathcal{K}'} \Rightarrow c}$$

Fig. 5.9: Traduction des expressions faisant intervenir les classes

Correction

Une expression typée dans un environnement Γ est traduite en une expression typable dans l'environnement $\overline{\Gamma}$ défini comme l'environnement Γ où chaque liaison de classe $c : (\mathcal{K}, \tau)$ est remplacé par une liaison $c : \tau$. Nous avons prouvé la correction de cette traduction :

Théorème 29 (Correction de la traduction) *Si* $\Gamma \vdash a : \tau \Rightarrow \overline{a}$, *alors* $\overline{\Gamma} \vdash \overline{a} : \tau$.

La preuve est facile. En effet, la plupart des constructions du langage de classe ont une contrepartie directe dans le langage de base.

Démonstration. Par induction sur les règles de traduction.

Nous montrons simultanément que si $\Gamma \vdash \mathit{expr} : (\mathcal{K}, \tau) \Rightarrow \overline{a}$ alors $\overline{\Gamma} \vdash \overline{a} : \tau$, ainsi que les résultats similaires concernant les autres parties de la syntaxe des classes. Nous avons omis les cas ne faisant intervenir que des constructions du calcul de base car ils sont immédiats.

–
$$\frac{}{\Gamma \vdash \emptyset : (\emptyset, \zeta(\alpha)[\emptyset]^{\emptyset}_{\emptyset}) \Rightarrow \zeta(x : \alpha)[\emptyset : \emptyset]^{\emptyset}_{\emptyset}} \ (\textsc{Classe-Vide})$$

Clairement,
$$\frac{\ldots}{\overline{\Gamma} \vdash \zeta(x : \alpha)[\emptyset : \emptyset]^{\emptyset}_{\emptyset} : \zeta(\alpha)[\emptyset]^{\emptyset}_{\emptyset}} \ (\textsc{Proto})$$

–
$$\frac{\begin{array}{c}(c : (\mathcal{K}, \tau' \to \tau)) \in \Gamma \\ \Gamma \vdash a : \tau'_1 \Rightarrow \overline{a} \qquad \Gamma \vdash \tau'_1 \preceq \tau'\end{array}}{\Gamma \vdash \texttt{inherit}\ c(a) : (\mathcal{K}, \tau) \Rightarrow c(\overline{a})} \ (\textsc{Héritage})$$

Alors,
$$\frac{\dfrac{(c : \tau' \to \tau) \in \overline{\Gamma}}{\overline{\Gamma} \vdash c : \tau' \to \tau}\ (\textsc{Var}) \qquad \dfrac{\overline{\Gamma} \vdash \overline{a} : \tau'_1 \qquad \overline{\Gamma} \vdash \tau'_1 \preceq \tau'}{\overline{\Gamma} \vdash \overline{a} : \tau'}\ (\textsc{Sub})}{\overline{\Gamma} \vdash c(\overline{a}) : \tau} \ (\textsc{App})$$

–
$$\frac{\Gamma \vdash \mathit{extension} : (\mathcal{K}', \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}) \Rightarrow a}{\begin{array}{l}\Gamma \vdash (\mathit{extension}\ \texttt{abstract}\ l : \varsigma(\alpha)\tau) : \\ (\mathcal{K}', \zeta(\alpha)[\mathcal{L}; (l : \tau)]^{\mathcal{A} \cup \{l\}}_{\mathcal{K}}) \Rightarrow a + (l : \varsigma(\alpha)\tau)\end{array}} \ (\textsc{Méthode-Abstraite})$$

Alors,
$$\frac{\overline{\Gamma} \vdash a : \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}}{\overline{\Gamma} \vdash a + (l : \varsigma(\alpha)\tau) : \zeta(\alpha)[\mathcal{L}; (l : \tau)]^{\mathcal{A} \cup \{l\}}_{\mathcal{K}}} \ (\textsc{Extension})$$

–
$$\frac{\Gamma \vdash \mathit{extension} : (\mathcal{K}', \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}) \Rightarrow a}{\begin{array}{l}\Gamma, (k : \zeta(\alpha)[\mathcal{L}]) \vdash \mathit{extension} : \\ (\mathcal{K}' \cup \{k\}, \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K} \cup \{k\}}) \Rightarrow \langle a \rangle_k\end{array}} \ (\textsc{Vue-Locale})$$

On suppose $\overline{\Gamma} \vdash a : \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}$. Alors, d'après le lemme 19,
$$\overline{\Gamma}; (k : \zeta(\alpha)[\mathcal{L}]) \vdash a : \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}$$

D'où

$$\frac{\begin{array}{c}\overline{\Gamma}; (k : \zeta(\alpha)[\mathcal{L}]) \vdash a : \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}} \\ (k : \zeta(\alpha)[\mathcal{L}]) \in (\overline{\Gamma}; (k : \zeta(\alpha)[\mathcal{L}]))\end{array}}{\overline{\Gamma}; (k : \zeta(\alpha)[\mathcal{L}]) \vdash \langle a \rangle_k : \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}\cup\{k\}}} \text{ (VUE-CAPTURE)}$$

–

$$\frac{\begin{array}{c}\Gamma, (k : \zeta(\alpha)[\mathcal{L}]) \vdash \mathit{corps} : (\mathcal{K}', \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}) \Rightarrow a' \\ \Gamma; (k : \zeta(\alpha)[\mathcal{L}]); (\alpha <\# \mathcal{K}); (x : \alpha) \vdash a : \mathcal{L}(l) \Rightarrow \overline{a}\end{array}}{\begin{array}{c}\Gamma, (k : \zeta(\alpha)[\mathcal{L}]) \vdash \mathit{corps}\ \texttt{method}\ l = \varsigma(x : \alpha)a : \\ (\mathcal{K}', \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}\setminus\{l\}}_{\mathcal{K}}) \Rightarrow a'.l \Leftarrow \varsigma(x : \alpha)\overline{a}\end{array}} \text{ (MÉTHODE)}$$

Alors,

$$\frac{\begin{array}{c}\overline{\Gamma}; (k : \zeta(\alpha)[\mathcal{L}]) \vdash a' : \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}} \\ \overline{\Gamma}; (k : \zeta(\alpha)[\mathcal{L}]); (\alpha <\# \mathcal{K}); (x : \alpha) \vdash \overline{a} : \mathcal{L}(l)\end{array}}{\overline{\Gamma}; (k : \zeta(\alpha)[\mathcal{L}]) \vdash a'.l \Leftarrow \varsigma(x : \alpha)\overline{a} : \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}\setminus\{l\}}_{\mathcal{K}}} \text{ (REDÉFINITION)}$$

–

$$\frac{\Gamma, (k : \zeta(\alpha)[\mathcal{L}]) \vdash \mathit{corps} : (\mathcal{K}, \tau) \Rightarrow a}{\Gamma, (k : \zeta(\alpha)[\mathcal{L}]) \vdash \texttt{object}\ \mathit{corps}\ \texttt{end} : (\mathcal{K}, \tau)\tau \Rightarrow a} \text{ (CLASSE-CORPS)}$$

Clair.

–

$$\frac{\begin{array}{c}\Gamma, (k : \zeta(\alpha)[\mathcal{L}]) \vdash \mathit{expr} : (\mathcal{K}', \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}) \Rightarrow a \\ \Gamma \vdash \mathit{type} : \zeta(\alpha)[\mathcal{L}_1]^{\mathcal{A}}_{\mathcal{K}_1} \qquad \mathcal{L}_1 \subseteq \mathcal{L} \\ \mathcal{K}_1 \subseteq \mathcal{K} \qquad \mathcal{A} \subseteq \operatorname{dom} \mathcal{L}_1\end{array}}{\begin{array}{c}\Gamma, (k : \zeta(\alpha)[\mathcal{L}]) \vdash (\mathit{expr} : \mathit{type}) : \\ (\mathcal{K} \setminus \mathcal{K}_1, \zeta(\alpha)[\mathcal{L}_1]^{\mathcal{A}}_{\mathcal{K}}) \Rightarrow a \setminus (\operatorname{dom} \mathcal{L} \setminus \operatorname{dom} \mathcal{L}_1)\end{array}} \text{ (CLASSE-CONTRAINTE)}$$

Alors, $\operatorname{dom} \mathcal{L} \setminus \operatorname{dom} \mathcal{L}_1 \subseteq \operatorname{dom} \mathcal{L} \setminus \mathcal{A}$ et $\mathcal{L}|_{\operatorname{dom} \mathcal{L} \setminus (\operatorname{dom} \mathcal{L} \setminus \operatorname{dom} \mathcal{L}_1)} = \mathcal{L}_1$. Par conséquent,

$$\frac{\begin{array}{c}\overline{\Gamma}; (k : \zeta(\alpha)[\mathcal{L}]) \vdash a : \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}} \\ \operatorname{dom} \mathcal{L} \setminus \operatorname{dom} \mathcal{L}_1 \subseteq \operatorname{dom} \mathcal{L} \setminus \mathcal{A}\end{array}}{\overline{\Gamma}; (k : \zeta(\alpha)[\mathcal{L}]) \vdash a \setminus (\operatorname{dom} \mathcal{L} \setminus \operatorname{dom} \mathcal{L}_1) : \zeta(\alpha)[\mathcal{L}_1]^{\mathcal{A}}_{\mathcal{K}}} \text{ (RESTRICTION)}$$

–

$$\frac{\begin{array}{c}\Gamma; (x : \tau'), (k : \zeta(\alpha)[\mathcal{L}]) \vdash \mathit{expr} : (\mathcal{K}', \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}}) \Rightarrow a' \\ \Gamma; (k); (c : (\{k\}, \tau' \to \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}\setminus\mathcal{K}'\cup\{k\}})) \vdash a : \tau \Rightarrow \overline{a}\end{array}}{\begin{array}{c}\Gamma \vdash \texttt{class}\ c\,[k](x : \tau') = \mathit{expr}\ \texttt{in}\ a : \tau \Rightarrow \\ \texttt{open}\ (\rho(k : \zeta(\alpha)[\mathcal{L}])\texttt{pack}\ \lambda(x : \tau')a' \\ \texttt{as}\ \exists(k')(\tau' \to \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}\setminus\mathcal{K}'\cup\{k'\}}) \\ \texttt{hiding}\ \mathcal{K}')\ \texttt{as}\ [k, c]\ \texttt{in}\ \overline{a}\end{array}} \text{ (CLASSE-PRIVÉE)}$$

On peut supposer

$$\left\{\begin{array}{l}\overline{\Gamma}; (x : \tau'); (k : \zeta(\alpha)[\mathcal{L}]) \vdash a' : \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}} \\ \overline{\Gamma}; (k); (c : \tau' \to \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}\setminus\mathcal{K}'\cup\{k\}}) \vdash \overline{a} : \tau\end{array}\right.$$

D'où

$$\dfrac{\overline{\Gamma};(x:\tau');(k:\zeta(\alpha)[\mathcal{L}])\vdash a':\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}}}{\overline{\Gamma};(k:\zeta(\alpha)[\mathcal{L}])\vdash \lambda(x:\tau')a':\tau'\to\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}}}\ (\textsc{Abs})$$

On a $\mathcal{K}'\subseteq\mathcal{K}$. et donc $(\mathcal{K}\setminus\mathcal{K}'\cup\{k'\})\{\mathcal{K}'/k'\}=\mathcal{K}$. Soit $\tau_0=\exists(k')(\tau'\to\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}\setminus\mathcal{K}'\cup\{k'\}})$. Alors,

$$\dfrac{\dfrac{\overline{\Gamma};(k:\zeta(\alpha)[\mathcal{L}])\vdash \lambda(x:\tau')a':\tau'\to\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}}}{\overline{\Gamma};(k:\zeta(\alpha)[\mathcal{L}])\vdash \texttt{pack}\ \lambda(x:\tau')a'\ \texttt{as}\ \tau_0\ \texttt{hiding}\ \mathcal{K}':\tau_0}\ (\textsc{Pack})}{\overline{\Gamma}\vdash\rho(k:\zeta(\alpha)[\mathcal{L}])\texttt{pack}\ \lambda(x:\tau')a'\ \texttt{as}\ \tau_0\ \texttt{hiding}\ \mathcal{K}':\tau_0}\ (\textsc{Vue-Abs})$$

Finalement,

$$\dfrac{\begin{array}{c}\overline{\Gamma}\vdash\rho(k:\zeta(\alpha)[\mathcal{L}])\texttt{pack}\ \lambda(x:\tau')a'\ \texttt{as}\ \tau_0\ \texttt{hiding}\ \mathcal{K}':\tau_0\\ \overline{\Gamma};(k);(c:\tau'\to\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}\setminus\mathcal{K}'\cup\{k\}})\vdash\overline{a}:\tau\end{array}}{\begin{array}{c}\overline{\Gamma}\vdash\texttt{open}\ (\rho(k:\zeta(\alpha)[\mathcal{L}])\texttt{pack}\ \lambda(x:\tau')a'\ \texttt{as}\ \tau_0\\ \texttt{hiding}\ \mathcal{K}')\ \texttt{as}\ [k,c]\ \texttt{in}\ \overline{a}:\tau\end{array}}\ (\textsc{Open})$$

–

$$\dfrac{\begin{array}{c}\Gamma;(x:\tau'),(k:\zeta(\alpha)[\mathcal{L}])\vdash \mathit{expr}:(\mathcal{K}',\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}})\Rightarrow a'\\ \Gamma;(k);(c:(\emptyset,\tau'\to\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}\setminus\mathcal{K}'\cup\{k\}}))\vdash a:\tau\Rightarrow\overline{a}\end{array}}{\begin{array}{l}\Gamma\vdash\texttt{class}\ c\ [\texttt{protected}\ k](x:\tau')=\mathit{expr}\ \texttt{in}\ a:\tau\Rightarrow\\ \quad\texttt{open}\ (\rho(k:\zeta(\alpha)[\mathcal{L}])\texttt{pack}\ \lambda(x:\tau')a'\\ \quad\quad\texttt{as}\ \exists(k')(\tau'\to\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}\setminus\mathcal{K}'\cup\{k'\}})\\ \quad\quad\quad\texttt{hiding}\ \mathcal{K}')\ \texttt{as}\ [k,c]\ \texttt{in}\ \overline{a}\end{array}}\ (\textsc{Classe-Protégée})$$

La preuve est similaire à celle du cas précédent.

– Règle Classe-Publique

$$\dfrac{\begin{array}{c}\Gamma;(x:\tau'),(k:\zeta(\alpha)[\mathcal{L}])\vdash \mathit{expr}:(\mathcal{K}',\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}})\Rightarrow a'\\ \Gamma;(k'');(k':\zeta(\alpha)[\mathcal{L}']);(c:(\emptyset,\tau'\to\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}\setminus\mathcal{K}'\cup\{k',k''\}}))\vdash a\{k'/k\}:\tau\Rightarrow\overline{a}\end{array}}{\begin{array}{c}\Gamma\vdash\texttt{class}\ c\ [\texttt{public}\ k](x:\tau')=\mathit{expr}\ \texttt{in}\ a:\tau\Rightarrow\\ \rho(k':\zeta(\alpha)[\mathcal{L}'])\texttt{open}\ (\rho(k:\zeta(\alpha)[\mathcal{L}])\texttt{pack}\ \lambda(x:\tau')\langle a'\rangle_{k'}\\ \texttt{as}\ \exists(k_0)(\tau'\to\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}\setminus\mathcal{K}'\cup\{k_0,k'\}})\ \texttt{hiding}\ \mathcal{K}')\ \texttt{as}\ [k'',c]\ \texttt{in}\ \overline{a}\end{array}}$$

On peut supposer

$$\left\{\begin{array}{l}\overline{\Gamma};(x:\tau');(k:\zeta(\alpha)[\mathcal{L}])\vdash a':\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}}\\ \overline{\Gamma};(k'');(k':\zeta(\alpha)[\mathcal{L}']);(c:\tau'\to\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}\setminus\mathcal{K}'\cup\{k',k''\}})\vdash\overline{a}:\tau\end{array}\right.$$

D'où

$$\dfrac{\dfrac{\begin{array}{c}\overline{\Gamma};(x:\tau');(k:\zeta(\alpha)[\mathcal{L}])\vdash a:\zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}\\ (k:\zeta(\alpha)[\mathcal{L}])\in\overline{\Gamma};(x:\tau');(k:\zeta(\alpha)[\mathcal{L}])\end{array}}{\overline{\Gamma};(x:\tau');(k:\zeta(\alpha)[\mathcal{L}])\vdash\langle a'\rangle_k:\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}\cup\{k\}}}\ (\textsc{Capture})}{\overline{\Gamma};(k:\zeta(\alpha)[\mathcal{L}])\vdash\lambda(x:\tau')\langle a'\rangle_k:\tau'\to\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}\cup\{k\}}}\ (\textsc{Abs})$$

On a $\mathcal{K}' \subseteq \mathcal{K}$. et donc $(\mathcal{K} \setminus \mathcal{K}' \cup \{k_0, k'\})\{\mathcal{K}'/k_0\} = \mathcal{K} \cup \{k'\}$. Soit $\tau_0 = \exists(k_0)(\tau' \to \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}\setminus\mathcal{K}'\cup\{k_0,k'\}})$. Alors,

$$\dfrac{\dfrac{\overline{\Gamma}; (k : \zeta(\alpha)[\mathcal{L}]) \vdash \lambda(x : \tau')\langle a'\rangle_k : \tau' \to \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}\cup\{k\}}}{\overline{\Gamma}; (k : \zeta(\alpha)[\mathcal{L}]) \vdash \texttt{pack } \lambda(x : \tau')\langle a'\rangle_k \texttt{ as } \tau_0 \texttt{ hiding } \mathcal{K}' : \tau_0}\ (\textsc{Pack})}{\overline{\Gamma} \vdash \rho(k : \zeta(\alpha)[\mathcal{L}])\texttt{pack } \lambda(x : \tau')\langle a'\rangle_k \texttt{ as } \tau_0 \texttt{ hiding } \mathcal{K}' : \tau_0}\ (\textsc{Vue-Abs})$$

D'après le lemme 19,

$$\overline{\Gamma}; (k' : \zeta(\alpha)[\mathcal{L}']) \vdash \rho(k : \zeta(\alpha)[\mathcal{L}])\texttt{pack } \lambda(x : \tau')\langle a'\rangle_k \texttt{ as } \tau_0 \texttt{ hiding } \mathcal{K}' : \tau_0$$

D'où (règle Open) :

$$\dfrac{\begin{array}{c}\overline{\Gamma}; (k' : \zeta(\alpha)[\mathcal{L}']) \vdash \rho(k : \zeta(\alpha)[\mathcal{L}])\texttt{pack } \lambda(x : \tau')a' \texttt{ as } \tau_0 \texttt{ hiding } \mathcal{K}' : \tau_0 \\ \overline{\Gamma}; (k' : \zeta(\alpha)[\mathcal{L}']); (k''); (c : \tau' \to \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}\setminus\mathcal{K}'\cup\{k',k''\}}) \vdash \overline{a} : \tau\end{array}}{\begin{array}{c}\overline{\Gamma}; (k' : \zeta(\alpha)[\mathcal{L}']) \vdash \texttt{open } (\rho(k : \zeta(\alpha)[\mathcal{L}])\texttt{pack } \lambda(x : \tau')a' \texttt{ as } \tau_0 \\ \texttt{hiding } \mathcal{K}') \texttt{ as } [k'', c] \texttt{ in } \overline{a} : \tau\end{array}}$$

Finalement,

$$\dfrac{\begin{array}{c}\overline{\Gamma}; (k' : \zeta(\alpha)[\mathcal{L}']) \vdash \texttt{open } (\rho(k : \zeta(\alpha)[\mathcal{L}])\texttt{pack } \lambda(x : \tau')a' \texttt{ as } \tau_0 \\ \texttt{hiding } \mathcal{K}') \texttt{ as } [k'', c] \texttt{ in } \overline{a} : \tau\end{array}}{\begin{array}{c}\overline{\Gamma} \vdash \rho(k' : \zeta(\alpha)[\mathcal{L}'])\texttt{open } (\rho(k : \zeta(\alpha)[\mathcal{L}])\texttt{pack } \lambda(x : \tau')a' \texttt{ as } \tau_0 \\ \texttt{hiding } \mathcal{K}') \texttt{ as } [k'', c] \texttt{ in } \overline{a} : \tau\end{array}}\ (\textsc{Vue-Abs})$$

–

$$\dfrac{\begin{array}{c}(c : (\mathcal{K}', \tau' \to \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}})) \in \Gamma \\ \Gamma \vdash \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}} \preceq \zeta(\alpha)[\mathcal{L}]_{\mathcal{K}\setminus\mathcal{K}'}\end{array}}{\Gamma \vdash \texttt{new } c : \tau' \to \zeta(\alpha)[\mathcal{L}]_{\mathcal{K}\setminus\mathcal{K}'} \Rightarrow c}\ (\textsc{New})$$

Alors,

$$\dfrac{(c : \tau' \to \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}) \in \overline{\Gamma}}{\overline{\Gamma} \vdash c : \tau' \to \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}}\ (\textsc{Var})$$

et

$$\dfrac{\overline{\Gamma} \vdash \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}} \preceq \zeta(\alpha)[\mathcal{L}]_{\mathcal{K}\setminus\mathcal{K}'} \qquad \dfrac{}{\overline{\Gamma} \vdash \tau' \preceq \tau'}\ (\textsc{Sub-Réfl})}{\overline{\Gamma} \vdash \tau' \to \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}} \preceq \tau' \to \zeta(\alpha)[\mathcal{L}]_{\mathcal{K}\setminus\mathcal{K}'}}\ (\textsc{Sub-Flèche})$$

D'où

$$\dfrac{\begin{array}{c}\overline{\Gamma} \vdash c : \tau' \to \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}} \\ \overline{\Gamma} \vdash \tau' \to \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}} \preceq \tau' \to \zeta(\alpha)[\mathcal{L}]_{\mathcal{K}\setminus\mathcal{K}'}\end{array}}{\overline{\Gamma} \vdash c : \tau' \to \zeta(\alpha)[\mathcal{L}]_{\mathcal{K}\setminus\mathcal{K}'}}\ (\textsc{Sub})$$

■

Exemple

Nous illustrons cette traduction sur la classe `comparable`. C'est la même classe que celle de la section 5.1.2, dont le sucre syntaxique a été expansé.

$$
\begin{array}{l}
\texttt{class comparable}\ [\texttt{protected}\ k](\texttt{x0} : \texttt{int}) = \texttt{object} \\
\quad \texttt{abstract x} : \varsigma(\alpha)\texttt{int} \\
\quad \texttt{abstract égale} : \varsigma(\alpha)\alpha \to \texttt{bool} \\
\quad \texttt{method x} = \varsigma(\texttt{moi} : \alpha)\texttt{x0} \\
\quad \texttt{method égale} = \varsigma(\texttt{moi} : \alpha)\lambda(\texttt{autre} : \alpha)\texttt{moi}._k\texttt{x} = \texttt{autre}._k\texttt{x} \\
\texttt{end}
\end{array}
$$

La classe est codée comme une fonction prenant un argument `x0` et retournant un objet. Cet objet est construit à partir de l'objet vide. Tout d'abord, les méthodes de la classe sont allouées (cela correspond aux clauses `abstract`). Puis la vue k est capturée et les méthodes sont définies. La vue est utilisée par la méthode `égale` pour accéder aux autres méthodes de la classe. La vue k est rendue abstraite. La construction `open` rend finalement la classe et sa vue accessible au reste du programme.

$$
\begin{array}{l}
\texttt{open} \\
\quad \rho(k : \zeta(\alpha)[\texttt{x} : \texttt{int}; \texttt{égale} : \alpha \to \texttt{bool}]) \\
\qquad \texttt{pack} \\
\qquad\quad \lambda(\texttt{x0} : \texttt{int}) \\
\qquad\qquad \langle \zeta(\texttt{s} : \alpha)[\emptyset : \emptyset]^{\emptyset}_{\emptyset} \\
\qquad\qquad\quad + (\texttt{x} : \varsigma(\alpha)\texttt{int}) \\
\qquad\qquad\quad + (\texttt{égale} : \varsigma(\alpha)\alpha \to \texttt{bool})\rangle_k \\
\qquad\qquad\quad .\texttt{x} \Leftarrow \varsigma(\texttt{s} : \alpha)\texttt{x0} \\
\qquad\qquad\quad .\texttt{égale} \Leftarrow \varsigma(\texttt{s} : \alpha)\lambda(\texttt{autre} : \alpha)\texttt{s}._k\texttt{x} = \texttt{autre}._k\texttt{x} \\
\qquad \texttt{as}\ \exists(k')\texttt{int} \to \zeta(\alpha)[\texttt{get\_x} : \texttt{int}; \texttt{égale} : \alpha \to \texttt{bool}]^{\emptyset}_{\{k'\}}\ \texttt{hiding}\ \{k\} \\
\texttt{as}\ [k, \texttt{comparable}]\ \texttt{in} \ldots
\end{array}
$$

5.3.3 Fonctions amies

Une fonction amie est une fonction qui a un accès privilégié aux objets d'une classe. L'existence d'un tel accès privilégié est particulièrement important si l'on cherche à réduire l'interface publique des classes. En effet, si par exemple deux objets doivent interagir entre eux, les méthodes nécessaires à cette interaction devraient autrement être publiques. Une technique usuelle permettant d'encoder les fonctions amies est d'utiliser une méthode `repr` qui retourne l'état interne de l'objet [21]. Afin que seules les fonctions amies puissent accéder à cet état, le type de la méthode est caché en utilisant un type existentiel. Cette technique s'oppose à notre but, qui est de pouvoir masquer n'importe quelle méthode. En effet, si cette méthode est masquée dans une sous-classe, la fonction amie ne pourra pas manipuler les objets de cette sous-classe. Une autre possibilité est de placer la classe et la fonction amie dans un module n'exportant pas la classe mais seulement un constructeur pour les objets de cette classe [14]. Il est alors possible de cacher n'importe quelle méthode de la classe dans le type du constructeur en rendant ce type partiellement abstrait. Cette méthode empêche cependant de créer des sous-classes. Aucune de ces solutions n'est donc tout à fait satisfaisante. Nous allons montrer comment les vues publiques permettent de définir des fonctions amies sans cette limitation.

La classe `c` définie ci-dessous introduit une vue publique. Cette vue est utilisée par la fonction `f` pour accéder à la méthode `x` d'un objet possédant cette vue. La vue est rendue abstraite et la méthode `x` est masquée par une signature. Il est cependant toujours possible d'appliquer la fonction `f` à un objet de la classe `c`, bien que celle-ci ne montre aucune méthode. De plus, la classe `c` peut être librement étendue par héritage et la fonction `f` peut être appliquée aux objets des sous-classes, aussi longtemps que la vue abstraite est héritée.

```
module M : sig
  class c [protected k] : object end
  val f : c -> int
end = struct
  class c [public k] (x0 : int) = object
    method x : int = x0
  end
  let f (v : Obj('a)[ | k ]) = obj.[k]x
end
```

Nous n'avons pas formalisé de système de module, mais nous donnons une traduction possible de l'exemple dans le calcul de base [1].

La vue k est la vue publique de la classe `c`. Elle est introduite juste avant la définition de la structure. La classe et la fonction sont ensuite définies. Plutôt que d'introduire une vue locale pour la classe `c`, la vue k est réutilisée. Le module est représenté par un objet. Les valeurs de ses méthodes sont les composants de la structure, coercés vers leurs types définitifs. La classe doit être enveloppée dans une fonction de coercion afin de pouvoir masquer la méthode `x`.

$$
\begin{array}{l}
\texttt{open}\\
\quad \rho(k : \zeta(\alpha)[\mathtt{x} : \mathtt{int}])\\
\quad\quad \texttt{pack}\\
\quad\quad\quad \texttt{let c} = \lambda(\mathtt{x}_0 : \mathtt{int})\langle \zeta(\mathtt{s} : \alpha)[\emptyset : \emptyset]^{\emptyset}_{\emptyset} + (\mathtt{x} : \varsigma(\alpha)\mathtt{int})\rangle_k.\mathtt{x} \Leftarrow \varsigma(\mathtt{s} : \alpha)\mathtt{x}_0 \ \texttt{in}\\
\quad\quad\quad \texttt{let } f = \lambda(\mathtt{v} : \zeta(\alpha)[\emptyset]_{\{k\}})\mathtt{v}._k\mathtt{x} \ \texttt{in}\\
\quad\quad\quad \zeta(\mathtt{s'} : \alpha')[\emptyset : \emptyset]^{\emptyset}_{\emptyset}\\
\quad\quad\quad\quad + (\mathtt{c} : \varsigma(\alpha')\zeta(\alpha)[\emptyset]^{\emptyset}_{\{k\}}).\mathtt{c} \Leftarrow \varsigma(\mathtt{s}' : \alpha')\lambda(\mathtt{x}_1 : \mathtt{int})(\mathtt{c}(\mathtt{x}_1) \setminus \mathtt{x})\\
\quad\quad\quad\quad + (\mathtt{f} : \varsigma(\alpha')\zeta(\alpha)[\emptyset]_{\{k\}} \to \mathtt{int}).\mathtt{f} \Leftarrow \varsigma(\mathtt{s}' : \alpha')\mathtt{f}\\
\quad\quad \texttt{as } \exists(k')\zeta(\alpha')[\mathtt{c} : \zeta(\alpha)[\emptyset]^{\emptyset}_{\{k'\}} ; \mathtt{f} : \zeta(\alpha)[\emptyset]_{\{k'\}} \to \mathtt{int}]^{\emptyset}_{\emptyset} \ \texttt{hiding } \{k\}\\
\texttt{as } [k, \mathtt{m}] \ \texttt{in} \ldots
\end{array}
$$

5.4 À propos du typage

Le calcul a été conçu de manière à rendre la vérification de type facile.

Nous pensons que les règles de typage pourraient être aisément adaptées de manière à rendre la synthèse de type possible. On utiliserait pour cela des variables de rangée d'une manière similaire à Objective Caml. Le type d'un objet serait alors composé de deux rangées : une pour les méthodes et l'autre pour les vues. Les règles de filtrage ne seraient plus nécessaires. En effet, une contrainte $\alpha <\# \mathcal{K}$ signifie qu'il n'est pas possible d'accéder aux méthodes d'un

[1] Dans cette traduction, nous utilisons une construction `let` que nous n'avons pas formalisé dans le langage. On peut la considérer simplement comme du sucre syntaxique : $\texttt{let } x = a' \texttt{ in } a$ est l'expression a dans laquelle les occurences de x ont été remplacées par a'.

objet de type α via son interface principal, et que l'objet a les méthodes de $\mathcal{K}$ et éventuellement d'autres méthodes. C'est exactement ce qu'exprime un type d'objet dont la liste des méthodes est vide, et dont les vues sont une rangée extensible contenant les vues $\mathcal{K}$.

Afin de simplifier le calcul et les preuves de correction, nous n'avons autorisé que le sous-typage en largeur. L'ajout du sous-typage en profondeur ne devrait cependant pas poser de difficulté. Alors qu'à présent les variables de types apparaissant à l'intérieur d'un type d'objet sont non-variantes (ni covariantes, ni contravariantes), il serait alors correct d'étendre la définition de la covariance de façon à ce qu'une variable de type α apparaissant en position covariante dans un type d'objet si elle apparaît en position covariante dans le type de chaque méthode de l'objet (la définition de la contravariance peut être étendue de manière similaire).

5.5 Anomalie en présence d'héritage multiple

Alors que ce calcul peut être utilisé pour un langage à classes avec héritage simple, une anomalie rend problématique son utilisation pour un langage avec héritage multiple. En effet, l'abstraction fournie par la quantification existentielle des vues lorsqu'une même vue est capturée deux fois par le même objet peut être violée dans certains cas.

Considérons en effet un objet possédant une vue k. Cette vue peut être cachée du type de l'objet par une existentielle. On pourrait penser qu'il n'y a plus aucun moyen d'utiliser cette vue de l'objet depuis l'extérieur. En particulier, on s'attend à ce que les méthodes qui ne sont accessibles que par cette vue ne puissent pas être redéfinies. Cependant, si cette vue est toujours disponible dans l'environnement, il est possible de l'ajouter de nouveau à l'objet. Cela va modifier le dictionnaire associé à cette vue dans l'objet, et les méthodes peuvent alors être modifiées de nouveau.

Dans un langage avec héritage simple, la portée des vues peut être contrôlée de manière à éviter ce problème. Par contre, si une classe peut hériter deux fois d'une autre classe, elle héritera des vues de cette classe deux fois. Ainsi, si une vue a été masquée sur seulement l'un des chemins d'héritage, elle sera encore visible dans la classe.

Il n'est pas du tout évident d'éviter ce problème. Il faudrait certainement utiliser une sémantique complètement différente. Le calcul actuel fournit cependant un point de départ pour la recherche d'une telle sémantique, et il serait en particulier probablement possible de s'inspirer de ses règles de typage.

5.6 Conclusion

Ce chapitre montre que de bonnes propriétés de modularité peuvent être obtenues sans perte d'expressivité dans un langage à objets, et qu'en particulier il est possible de masquer n'importe quelles méthodes même en présence de méthodes binaires.

Les vues sont un ingrédient nécessaire à la correction de notre calcul. Mais nous avons présenté d'autres utilisations possibles des vues, en particulier pour définir des fonctions amies.

Il n'est pas évident de modifier le calcul présenté ici afin de l'utiliser comme base d'un langage à objets avec héritage multiple. Nous espérons cependant que cela soit possible, car de nombreuses variantes du calcul sont possibles.

Chapitre 6

Combiner modules et classes

Introduction

Les classes et les modules ont de nombreuses propriétés communes : ils offrent tous deux une forme d'abstraction ; ils permettent de structurer le code et facilitent sa réutilisation. À cause de ce recouvrement de fonctionnalités, il peut donc être parfois difficile de faire un choix entre les deux. Le problème se pose en particulier dans Objective Caml où classes et modules sont des contructions complètement indépendantes. Dans ce chapitre, nous présentons les bases théoriques d'un langage ne comportant qu'une construction pouvant être utilisée aussi bien comme classe que comme module. Ce résultat est obtenu en identifiant classes et modules. Certains composants des classes correspondent en effet à des composants des modules : par exemple, le corps des classes et les structures. D'autres composants normalement propres aux classes, comme les méthodes, seraient simplement ajoutés au langage de module. La plupart de ces opérations sont des extensions simples du langage de module. Une extension des systèmes de module existants, petite mais cependant plus significative, est également nécessaire : de même que les types ML sont quantifiés implicitements, les types des modules seraient implicitement quantifiés.

6.1 Les classes dans Objective ML

Le langage de classe que nous allons considérer n'est pas tout à fait celui présenté en section 2.3 page 57. Il est en fait plus proche de la formulation donnée dans [28]. Nous le décrivons donc rapidement ici.

Une classe simple est formée d'une *structure* contenant des variables d'instance et des méthodes.

$\texttt{class}\ \mathit{origin} = \texttt{struct val}\ x = 0; \texttt{method}\ \mathit{getx} = \varsigma(y)\ x\ \texttt{end}$

Cette classe a une variable d'instance x initialisée à 0 et une méthode *getx*. Quand cette méthode est appelée, la variable y est liée à l'objet auquel elle appartient. Ici, cet argument est en fait ignorée et la méthode retourne la valeur de la variable d'instance. Le type de cette classe est :

$$\forall(\alpha)\ \texttt{sig}\ (\langle\, \mathit{getx} : \mathit{int};\ \alpha\,\rangle)\ \texttt{val}\ x : \mathit{int}; \texttt{method}\ \mathit{getx}\ \texttt{end}$$

Le type entre parenthèse donne la forme du type d'un objet de cette classe. La variable de rangée α peut être instanciée dans une sous-classe, ce qui permet l'extension de la classe avec de nouvelles méthodes. Le reste de la signature de la classe indique que celle-ci a une variable x et que la méthode *getx* est définie.

Une classe peut-être définie comme une fonction prenant un paramètre d'initialisation et retournant une valeur de classe. Cela est montré dans la classe suivante :

$$\texttt{class}\ \mathit{genpoint} = \texttt{fun}\ x_0 \to \texttt{struct}\ \texttt{val}\ x = x_0;\ \texttt{method}\ \mathit{getx} = \varsigma(y)\ x\ \texttt{end}$$

Le type de cette classe est :

$$\forall(\alpha)\ \forall(\alpha')\ \big(\alpha \to \texttt{sig}\ (\langle\, \mathit{getx} : \alpha;\ \alpha' \,\rangle)\ \texttt{val}\ x : \alpha;\ \texttt{method}\ \mathit{getx}\ \texttt{end}\big)$$

L'expression de classe $\big(\mathit{genpoint}\ 1\big)$ est alors une classe dont la variable d'instance x a la valeur 1. L'opérateur `new` permet de créer un objet à partir d'une classe. Un objet contient une copie des variables d'instance et des méthodes de sa classe, mais seules ses méthodes peuvent être accédées directement. Cela est reflété par son type. Par exemple, $\big(\texttt{new}\ \mathit{genpoint}\ 1\big)$ est un objet de type $\langle\, \mathit{getx} : \mathit{int} \,\rangle$. La classe *genpoint* pourrait être également appliquée à un flottant plutôt qu'à un entier : c'est une classe polymorphe.

Les classes peuvent être étendues par héritage. Le composant `inherit` c d'une structure permet d'inclure tous les composants d'une classe parente c. La classe *cpoint* ci-dessous hérite la variable d'instance x et la méthode *getx* de son père *genpoint*, et définie une nouvelle méthode *color* :

$$\begin{array}{l}\texttt{class}\ \mathit{cpoint} = \texttt{fun}\ (c : \mathit{string}) \to \\ \quad \texttt{struct}\ \texttt{inherit}\ \mathit{genpoint}\ 1;\ \texttt{method}\ \mathit{color} = c\ \texttt{end}\end{array}$$

Le type de la classe *cpoint*, donné ci-dessous, montre la variable d'instance héritée et les deux méthodes :

$$\begin{array}{l}\forall(\alpha)\ \mathit{string} \to \\ \qquad\qquad \texttt{sig}\ (\langle\, \mathit{getx} : \mathit{int};\ \mathit{color} : \mathit{string};\ \alpha \,\rangle) \\ \qquad\qquad\quad \texttt{val}\ x : \mathit{int};\ \texttt{method}\ \mathit{getx};\ \texttt{method}\ \mathit{color} \\ \qquad\qquad \texttt{end}\end{array}$$

6.2 Modules à la ML

Nous présentons maintenant le système de modules que nous comparerons avec le langage de classe de la section précédente. Un module simple est composé d'une *structure* contenant plusieurs composants : définitions de types, de valeurs ou de sous-modules.

$$\begin{array}{l}\texttt{module}\ \mathit{intOrder} = \\ \quad \texttt{struct}\ \texttt{type}\ t = \mathit{int};\ \texttt{val}\ \mathit{less}\ (x : t)\ (y : t) = x < y\ \texttt{end}\end{array}$$

Un composant d'un module peut être accédé en utilisant la notation $m.x$: la fonction *intOrder.less* compare deux valeurs de type *intOrder.t*, ce type étant égal au type *int*.

Une *signature* de module liste le type de ses composants.

$$\texttt{sig}\ \texttt{type}\ t = \mathit{int};\ \texttt{val}\ \mathit{less} : t \to t \to \mathit{bool}\ \texttt{end}$$

Un module peut également être un *foncteur*, c'est-à-dire une fonction des modules dans les modules. Cette fonction peut être appliquée à n'importe quel module qui se conforme à une certaine interface, et retourne un nouveau module.

$$
\begin{array}{l}
\texttt{module}\ \mathit{listOrder} = \\
\quad \texttt{functor}\ (\mathit{ord} : \texttt{sig} \\
\qquad\qquad\qquad\quad \texttt{type}\ t;\ \texttt{val}\ \mathit{less} : t \to t \to \mathit{bool} \\
\qquad\qquad\qquad \texttt{end}) \to \\
\qquad \texttt{struct} \\
\qquad\quad \texttt{type}\ t = \mathit{ord}.t\ \mathit{list};\ \texttt{val}\ \mathit{less}\ (l1 : t)\ (l2 : t) = \ldots \\
\qquad \texttt{end}
\end{array}
$$

La signature de ce foncteur est

$$
\begin{array}{l}
\texttt{functor}\ (\mathit{ord} : \texttt{sig} \\
\qquad\qquad\quad \texttt{type}\ t;\ \texttt{val}\ \mathit{less} : t \to t \to \mathit{bool} \\
\qquad\qquad \texttt{end}) \to \\
\quad \texttt{sig} \\
\qquad \texttt{type}\ t = \mathit{ord}.t\ \mathit{list};\ \texttt{val}\ \mathit{less} : t \to t \to \mathit{bool} \\
\quad \texttt{end}
\end{array}
$$

Ce foncteur peut être appliqué au module précédent, produisant un module de type :

$$\texttt{sig type}\ t = \mathit{intOrder}.t\ \mathit{list};\ \texttt{val}\ \mathit{less} : t \to t \to \mathit{bool}\ \texttt{end}$$

Un module peut étendre un autre module (la construction `open` a le même sens qu'en SML : les composants du module ouvert sont liées dans le module courant).

$$
\begin{array}{l}
\texttt{module}\ \mathit{intOrder2} = \\
\quad \texttt{struct} \\
\qquad \texttt{open}\ \mathit{intOrder}; \\
\qquad \texttt{val}\ \mathit{min}\ x\ y = \texttt{if}\ \mathit{less}\ x\ y\ \texttt{then}\ x\ \texttt{else}\ y \\
\quad \texttt{end}
\end{array}
$$

Le module *intOrder2* ci-dessus contient donc les mêmes composants que le module *intOrder*, plus la valeur *min*.

6.3 Comparaison entre classes et modules

Les deux sections précédentes ont présenté de petits langages de classes et de modules. Ces langages sont très similaires : ils utilisent tous deux des structures, qui peuvent être paramétrées. La figure 6.1 fournit une comparaison de la syntaxe des deux langages, mettant en valeur leurs similarités. Il y a bien sûr un certain nombre de différences, que nous étudierons dans le reste de cette section.

Ainsi, dans le langage de classe, les fonctions prennent pour argument des valeurs du langage de base. Par contre, les foncteurs prennent en argument des modules. Une autre différence est que certains composants des structures des classes n'ont pas de correspondance dans les modules, et vice versa. Les types

Expressions de classes et de modules

$c ::= z$	$m ::= p$	Accès à une valeur
$\mid \texttt{fun}\ z \to c$	$\mid \texttt{functor}\ (z : \phi) \to m$	Abstraction
$\mid c\ e$	$\mid m\ m$	Application
$\mid \texttt{struct}\ b\ \texttt{end}$	$\mid \texttt{struct}\ b\ \texttt{end}$	Structure
$b ::= \epsilon \mid d;\ b$	$b ::= \epsilon \mid d;\ b$	

Composants des structures

$d ::= \texttt{inherit}\ c$	$d ::= \texttt{open}\ m$	Héritage / inclusion
$\mid \texttt{val}\ x = e$	$\mid \texttt{val}\ x = e$	Valeur
$\mid \texttt{method}\ y = \varsigma(x)\ e$		Méthode
	$\mid \texttt{type}\ t = \tau$	Type
	$\mid \texttt{module}\ z = m$	Module
	$T ::= \tau \mid \forall(\alpha)\ T$	Schéma de type
$\Phi ::= \phi \mid \forall(\alpha)\ \Phi$		Schéma de type de classe

Types de classes et de modules

$\phi ::= \tau \to \phi$	$\phi ::= \texttt{functor}\ (z : \phi) \to \phi$	Type fonctionnel
$\mid \texttt{sig}\ (\tau)\ \chi\ \texttt{end}$	$\mid \texttt{sig}\ \chi\ \texttt{end}$	Type de signature
$\chi ::= \epsilon \mid \psi;\ \chi$	$\chi ::= \epsilon \mid \psi;\ \chi$	

Composants des signatures

$\psi ::= \texttt{val}\ x : \tau$	$\psi ::= \texttt{val}\ x : T$	Valeur
$\mid \texttt{method}\ m$		Méthode
	$\mid \texttt{type}\ t = \tau$	Type manifeste
	$\mid \texttt{type}\ t$	Type abstrait
	$\mid \texttt{module}\ z : \phi$	Module

FIG. 6.1: Comparaison entre un langage de classe et un langage de module

sont égalements différents. En effet, les types des classes peuvent être quantifiés et le type des composants des classes peut contenir des variables libres. Au contraire, tous les types apparaissant dans un type de module doivent être clos, et les types de modules n'ont donc pas besoin d'être quantifiés. Finalement, la signature d'une classe contient un composant distingué des autres, représentant le type d'une instance de la classe. Ces différences ne sont cependant pas incompatible, comme nous allons le voir dans la suite de cette section en les examinant plus en détail.

Une abstraction de classe peut être codée comme un foncteur : l'expression $\texttt{fun}\ x \to m$ serait traduite en l'expression

$$\texttt{functor}\ (z : \texttt{sig}\ \texttt{val}\ x : \tau\ \texttt{end}) \to m[z.x/x]$$

Cela nécessite cependant que le type τ soit connu, ce qui n'est généralement pas le cas (notamment si le type de x est polymorphe). Nous reviendrons sur ce point plus tard dans cette section.

Il paraît possible d'autoriser les structures à contenir aussi bien des composants de modules que des composants de classe. L'héritage (`inherit`) et l'inclusion de module (`open`) peuvent être fusionnés en une même opération : en effet, elles incluent toutes deux tous les composants d'une autre structure. Les valeurs (`val`) ont également une sémantique compatible.

La quantification des types de classe pose un problème plus sérieux. Il est bien possible de coder la quantification explicite dans un langage de modules [15] : $\forall(\alpha)\ m$ peut être réécrit en

$$\texttt{functor}\ (z : \texttt{sig}\ \texttt{type}\ t\ \texttt{end}) \to m[z.t/\alpha]$$

Cependant, pour supprimer la quantification, la valeur du type t doit être fournie explicitement. Cela ne s'intègre pas très bien avec un langage de base avec polymorphisme implicite comme ML. Par exemple, la création d'un objet de la classe *genpoint* (présenté dans la section 6.1 page 135) devrait être écrite de manière bien peu concise :

$$\mathit{genpoint}\ \big(\texttt{sig}\ \texttt{type}\ t = \mathit{int};\ \texttt{val}\ x = 1\ \texttt{end}\big)$$

Pour résoudre ce problème, nous proposons d'autoriser la quantification implicite des types de module, en utilisant les règles suivantes pour l'introduction et l'élimination de la quantification :

$$\frac{A \vdash m : \forall(\alpha)\ \Phi}{A \vdash m : \Phi[\tau/\alpha]} \qquad\qquad \frac{A \vdash m : \Phi \qquad \alpha \notin \mathrm{FV}(A)}{A \vdash m : \forall(\alpha)\ \Phi}$$

Il est alors intéressant de pouvoir contrôler la portée des variables de type en les liant explicitement :

$$\frac{A;\ \big(\texttt{var}\ \alpha = \tau\big) \vdash m : \Phi}{A \vdash \big([\alpha]\ m\big) : \Phi}$$

Lors du typage de l'expression de module m, la variable de type α se retrouve liée à un type τ. Alors, les variables libres de τ ne peuvent pas être généralisées dans m, car elles apparaissent dans l'environnement. Avec cette extension, l'expression $(\texttt{fun}\ x \to m)$ peut être codée ainsi :

$$[\alpha]\,\texttt{functor}\,(z : \texttt{sig}\ \texttt{val}\ x : \alpha\ \texttt{end}) \to m[z.x/x]$$

Les autres différences entre les deux langages sont mineures. En particulier, le type des valeurs n'est pas généralisé dans les signatures des classes, mais seulement parce que ce n'est pas utile.

6.4 Présentation du langage de modules étendu

Nous présentons maintenant le système de modules complet. C'est une variante de celui proposé par [18]. Dans cette variante, les noms apparaissant dans un programme ne sont pas annotés. Par contre, les noms apparaissant dans un type le sont afin d'éviter des ambiguités. Nous laissons le langage de base pour l'essentiel indéterminé.

6.4.1 Syntaxe

La syntaxe est présentée dans la figure 6.2. Les variables z, t, x et y sont des noms (de module, de type, de valeur et de composant de module respectivement). Les noms ne sont pas soumis à l'α-conversion. Les types sont notés à l'aide de lettres grecques. Les schémas de types sont notés par la lettre capitale correspondant à la minuscule du type associé. Les tildes ($\tilde{\ }$) au-dessus des types *syntaxiques* (qui peuvent apparaître dans une expression) permettent de les différencier des types *enrichis* (qui permettent de typer le langage).

Par rapport au système de module présenté par Xavier Leroy, un certain nombre d'additions ont été apportées à la syntaxe. Les types sont étendus avec des types récursifs et des types d'objet. L'expression de module $[\alpha]\ m$ introduit une variable α dans l'expression de module m. Le composant $\texttt{method}\ y = \varsigma(x)\ e$ permet de définir une méthode ; lors de l'appel de cette méthode, son corps e est évalué après avoir remplacé le nom x par l'objet auquel la méthode appartient. Le composent $\texttt{inherit}\ m$ inclut les composants de la structure m dans la structure courante. Les types de module sont quantifiés par des variables de types. Une signature $(\texttt{sig}\ (\tilde{\tau})\ \tilde{\chi}\ \texttt{end})$ est composée d'une suite de types de composants $\tilde{\chi}$, ainsi que d'un type $\tilde{\tau}$ représentant le type d'une instance du module (vu comme une classe). Enfin, le composant de signature $\texttt{method}\ y$ indique que le module définit une méthode y.

6.4.2 Règles de typage

Des types plus précis, présentés dans la figure 6.3, sont nécessaires pour le typage : les noms peuvent en effet être ambigus, et doivent donc être annotés afin de lever cette ambiguïté. Un nom annoté est appelé un *identificateur*. Ces annotations sont la seule différence entre les types syntaxiques décris précédemment et les types de cette figure. Les identificateurs sont soumis à l'α-conversion. Cependant, seule l'annotation peut être renommée. Les types sont considérés modulo renommage des identificateurs liés.

Un environnement est une séquence contenant des types de composant de module et des liaisons d'une variable de type α à un type τ. Un identificateur ne peut pas être lié deux fois dans un environnement (c'est-à-dire qu'un nom ne

$\tilde{p} ::= z \mid \tilde{p}.z$	Chemin d'accès
$e ::= x \mid \tilde{p}.x \mid \texttt{new}\ m$ $\quad \mid \texttt{fun}\ x \to e \mid \ldots$	Expression
$\tilde{T} ::= \tilde{\tau} \mid \forall(\alpha)\ \tilde{T}$	Schéma de type
$\tilde{\tau} ::= t \mid \tilde{p}.t \mid \alpha \mid \tilde{\tau} \to \tilde{\tau}$ $\quad \mid \mu(\alpha)\ \tilde{\tau} \mid \langle\, \tilde{\omega}\, \rangle$	Type de base
$\tilde{\omega} ::= \epsilon \mid \alpha \mid y : \tilde{\tau};\, \tilde{\omega}$	Type d'object
$m ::= \tilde{p} \mid \texttt{struct}\ b\ \texttt{end}$ $\quad \mid \texttt{functor}\ (z : \tilde{\Phi}) \to m$ $\quad \mid m\ m \mid (m : \tilde{\Phi}) \mid [\alpha]\ m$	Expression de module
$b ::= \epsilon \mid d;\, b$	Corps de module
$d ::= \texttt{val}\ x = e$ $\quad \mid \texttt{method}\ y = \varsigma(x)\ e$ $\quad \mid \texttt{module}\ z = m$ $\quad \mid \texttt{type}\ t = \tilde{\tau}$ $\quad \mid \texttt{inherit}\ m$	Composants de module
$\tilde{\Phi} ::= \tilde{\phi} \mid \forall(\alpha)\ \tilde{\Phi}$	Schéma de type de module
$\tilde{\phi} ::= \texttt{functor}\ (z : \tilde{\Phi}) \to \tilde{\Phi}$ $\quad \mid \texttt{sig}\ (\tilde{\tau})\ \tilde{\chi}\ \texttt{end}$	Type de module
$\tilde{\chi} ::= \epsilon \mid \tilde{\psi};\, \tilde{\chi}$	Type de corps de module
$\tilde{\psi} ::= \texttt{val}\ x : \tilde{T}$ $\quad \mid \texttt{method}\ y$ $\quad \mid \texttt{module}\ z : \tilde{\Phi}$ $\quad \mid \texttt{type}\ t \mid \texttt{type}\ t = \tilde{\tau}$	Type de composant de module

FIG. 6.2: Syntaxe

$p ::= z_i \mid p.z$	Chemin d'accès
$T ::= \tau \mid \forall(\alpha)\ T$	Schéma de type
$\tau ::= t_i \mid p.t \mid \alpha \mid \tau \to \tau$ $\mid \mu(\alpha)\ \tau \mid \langle\, \omega\, \rangle$	Type de base
$\omega ::= \epsilon \mid \alpha \mid y : \tau;\ \omega$	Type d'objet
$\Phi ::= \phi \mid \forall(\alpha)\ \Phi$	Schéma de type de module
$\phi ::= \texttt{functor}\ (z_i : \Phi) \to \Phi$ $\mid \texttt{sig}\ (\tau)\ \chi\ \texttt{end}$	Type de module
$\chi ::= \epsilon \mid \psi;\ \chi$	Type de corps de module
$\psi ::= \texttt{val}\ x_i : T \mid \texttt{method}\ y$ $\mid \texttt{module}\ z_i : \Phi$ $\mid \texttt{type}\ t_i \mid \texttt{type}\ t_i = \tau$	Type de composant de module

FIG. 6.3: Types

peut pas être lié deux fois avec la même annotation, mais il est possible de lier deux identificateurs de même nom s'ils ont des annotations distinctes).

$$A ::= \epsilon \mid A;\ \psi \mid A;\ \big(\texttt{var}\ \alpha_i = \tau\big)$$

La notation $A(n)$ représente la liaison la plus à droite d'un identificateur de nom n dans l'environnement A. Les variables libres d'un environnement A et d'un type τ sont notées respectivement $\mathrm{FV}(A)$ et $\mathrm{FV}(\tau)$. L'opération $A + \chi$ décharge le contenu d'un corps de module χ dans l'environnement A. Elle est définie par $A + \big(\psi;\ \chi\big) = \big(A;\ \psi\big) + \chi$ et $A + \epsilon = A$. L'opération $\phi +_A \phi'$ fusionne deux signatures de module ϕ et ϕ'. Elle est définie par $\big(\texttt{sig}\ (\tau)\ \chi\ \texttt{end}\big) +_A \big(\texttt{sig}\ (\tau')\ \chi'\ \texttt{end}\big) = \texttt{sig}\ (\tau')\ \chi + \chi'\ \texttt{end}$ quand $A \vdash \tau \approx \tau'$, où l'opération $\chi + \chi'$ est définie par $\big(\psi;\ \chi\big) + \chi' = \psi;\ \big(\chi + \chi'\big)$ et $\epsilon + \chi' = \chi'$.

Les règles de typage associent des types de modules aux expressions de modules ($A \vdash m : \Phi$). Elles utilisent une notion d'équivalence de types $A \vdash T \approx T'$ et une relation de sous-typage entre schémas de types de modules $A \vdash \Phi <: \Phi'$. Ces deux jugements sont standards (leur extension aux schémas de type excepté). Ils sont définis dans la figure 6.4. La relation $\approx$ est étendue aux types de modules ($A \vdash \Phi \approx \Phi'$) comme étant la relation d'équivalence associée au préordre $<:$. La traduction des types $A \vdash \tilde{T} : T$ et $A \vdash \tilde{\Phi} : \Phi$ est présentée en figure 6.5.

Afin de rendre les règles de typage plus lisibles, quelques hypothèses implicites sont faites systématiquement : tout type apparaissant dans une règle de

Chemins d'accès

$$\frac{\left(\texttt{module}\ z_i : \Phi\right) \in A}{A \vdash z_i : \Phi} \qquad \frac{A \vdash p : \left(\texttt{sig}\ (\tau)\ \chi\ \texttt{end}\right) \qquad \left(\texttt{module}\ z_i : \Phi\right) \in \chi \qquad A + \chi \vdash \Phi \approx \Phi'}{A \vdash p.z : \Phi'}$$

$$\frac{A \vdash p : \forall(\alpha)\ \Phi}{A \vdash p : \Phi[\tau/\alpha]}$$

Équivalence de type (les règles de congruence, réflexivité, symmétrie et transitivité sont omises)

$$\frac{\left(\texttt{type}\ t_i = \tau\right) \in A}{A \vdash t_i \approx \tau} \qquad \frac{A \vdash p : \texttt{sig}\ (\tau)\ \chi\ \texttt{end} \qquad \left(\texttt{type}\ t_i = \tau\right) \in \chi}{A \vdash p.t \approx \tau}$$

Sous-typage de module

$$\frac{A \vdash \tau_1 \approx \tau_2}{A \vdash \tau_1 <: \tau_2} \qquad \frac{A \vdash \Phi'_2 <: \Phi'_1 \qquad A; \left(\texttt{module}\ z_i : \Phi'_2\right) \vdash \Phi_1 <: \Phi_2}{A \vdash \texttt{functor}\ (z_i : \Phi'_1) \to \Phi_1 <: \texttt{functor}\ (z_i : \Phi'_2) \to \Phi_2}$$

$$\frac{A \vdash \tau_1 \approx \tau_2 \qquad \forall d_2 \in \chi_2 \cdot \exists d_1 \in \chi_1 \cdot A + \chi_1 \vdash d_1 <: d_2}{A \vdash (\texttt{sig}\ (\tau_1)\ \chi_1\ \texttt{end}) <: (\texttt{sig}\ (\tau_2)\ \chi_2\ \texttt{end})}$$

$$\frac{A \vdash T_1 <: T_2}{A \vdash (\texttt{val}\ x_i : T_1) <: (\texttt{val}\ x_i : T_2)}$$

$$\frac{A \vdash \Phi_1 <: \Phi_2}{A \vdash (\texttt{module}\ z_i : \Phi_1) <: (\texttt{module}\ z_i : \Phi_2)} \qquad \frac{}{A \vdash (\texttt{method}\ y) <: (\texttt{method}\ y)}$$

$$\frac{}{A \vdash (\texttt{type}\ t_i) <: (\texttt{type}\ t_i)} \qquad \frac{}{A \vdash (\texttt{type}\ t_i = \tau) <: (\texttt{type}\ t_i)}$$

$$\frac{A \vdash \tau \approx \tau'}{A \vdash (\texttt{type}\ t_i = \tau) <: (\texttt{type}\ t_i = \tau')} \qquad \frac{A \vdash t_i \approx \tau}{A \vdash (\texttt{type}\ t_i) <: (\texttt{type}\ t_i = \tau)}$$

$$\frac{A \vdash \Phi_1[\tau/\alpha] <: \Phi_2}{A \vdash \forall(\alpha)\ \Phi_1 <: \Phi_2} \qquad \frac{A \vdash \Phi_1 <: \Phi_2 \qquad \alpha \notin \mathrm{FV}(A)}{A \vdash \Phi_1 <: \forall(\alpha)\ \Phi_2}$$

$$\frac{A \vdash T_1[\tau/\alpha] <: T_2}{A \vdash \forall(\alpha)\ T_1 <: T_2} \qquad \frac{A \vdash T_1 <: T_2 \qquad \alpha \notin \mathrm{FV}(A)}{A \vdash T_1 <: \forall(\alpha)\ T_2}$$

Équivalence des schémas de types de modules

$$\frac{A \vdash \Phi_1 <: \Phi_2 \qquad A \vdash \Phi_2 <: \Phi_1}{A \vdash \Phi_1 \approx \Phi_2}$$

FIG. 6.4: Relations d'équivalence et de sous-typage

Types de base (les règles structurelles sont omises)

(Type-Name)
$$\frac{(\mathtt{type}\ t_i[=\tau\,]) \in A}{A \vdash t : t_i}$$

(Type-Projection)
$$\frac{A \vdash \tilde{p} : \mathtt{sig}\ (\tau)\ \chi\ \mathtt{end} \qquad (\mathtt{type}\ t_i = \tau) \in \chi}{A \vdash \tilde{p}.t : \tau}$$

(Type-Var)
$$\frac{A(\alpha) = (\mathtt{var}\ \alpha_i = \tau)}{A \vdash \alpha : \tau}$$

(Type-Quant)
$$\frac{A;\ (\mathtt{var}\ \alpha_i = \alpha') \vdash \tilde{T} : T \qquad \alpha' \notin \mathrm{FV}(A)}{A \vdash \forall(\alpha)\ \tilde{T} : \forall(\alpha')\ T}$$

Signature

(Sig-Empty)
$$\frac{}{A \vdash \epsilon : \epsilon}$$

(Sig-Then)
$$\frac{A \vdash \tilde{\psi} : \psi \qquad A;\ \psi \vdash \tilde{\chi} : \chi}{A \vdash (\tilde{\psi};\ \tilde{\chi}) : (\psi;\ \chi)}$$

(Sig-Value)
$$\frac{A \vdash \tilde{T} : T}{A \vdash (\mathtt{val}\ x : \tilde{T}) : (\mathtt{val}\ x_i = T)}$$

(Sig-Module)
$$\frac{A \vdash \tilde{\Phi} : \Phi}{A \vdash (\mathtt{module}\ z : \tilde{\Phi}) : (\mathtt{module}\ z_i : \Phi)}$$

(Sig-Manifest-Type)
$$\frac{A \vdash \tilde{\tau} : \tau \qquad \mathrm{FV}(\tau) = \emptyset}{A \vdash (\mathtt{type}\ t = \tilde{\tau}) : (\mathtt{type}\ t_i = \tau)}$$

(Sig-Abstract-Type)
$$\frac{}{A \vdash (\mathtt{type}\ t) : (\mathtt{type}\ t_i)}$$

(Sig-Method)
$$\frac{}{A \vdash (\mathtt{method}\ y) : (\mathtt{method}\ y)}$$

(Signature)
$$\frac{A \vdash \tau : \tau' \qquad A \vdash \chi : \chi'}{A \vdash (\mathtt{sig}\ (\tau)\ \chi\ \mathtt{end}) : (\mathtt{sig}\ (\tau')\ \chi'\ \mathtt{end})}$$

(Sig-Functor)
$$\frac{A \vdash \tilde{\Phi}' : \Phi' \qquad A;\ (\mathtt{module}\ z_i : \Phi') \vdash \tilde{\Phi} : \Phi}{A \vdash (\mathtt{functor}\ (z : \tilde{\Phi}') \to \tilde{\Phi}) : (\mathtt{functor}\ (z : \Phi') \to \Phi)}$$

(Sig-Quant)
$$\frac{A;\ (\mathtt{var}\ \alpha_i = \alpha') \vdash \tilde{\Phi} : \Phi \qquad \alpha' \notin \mathrm{FV}(A)}{A \vdash \forall(\alpha)\ \tilde{\Phi} : \forall(\alpha')\ \Phi}$$

Fig. 6.5: Traduction des types

Bonne formation des types ($A \vdash T$ type) (les règles structurelles sont omises)

$$\frac{(\texttt{type}\ t_i[=\tau\,])\in A}{A\vdash t_i\ \text{type}}\qquad\frac{A\vdash p:\texttt{sig}\ (\tau)\ \chi\ \texttt{end}\qquad(\texttt{type}\ t_i=\tau)\in\chi}{A\vdash p.t\ \text{type}}$$

Bonne formation des types de modules ($A \vdash \Phi$ module type)

$$\frac{}{A\vdash\epsilon\ \text{decl}}\qquad\frac{A\vdash T\ \text{type}\qquad A\vdash\chi\ \text{decl}}{A\vdash(\texttt{val}\ x_i=T;\chi)\ \text{decl}}\qquad\frac{A\vdash\chi\ \text{decl}}{A\vdash(\texttt{method}\ y;\chi)\ \text{decl}}$$

$$\frac{A\vdash\tau\ \text{type}\qquad A;(\texttt{type}\ t_i=\tau)\vdash\chi\ \text{decl}}{A\vdash(\texttt{type}\ t_i=\tau;\chi)\ \text{decl}}\qquad\frac{A;(\texttt{type}\ t_i)\vdash\chi\ \text{decl}}{A\vdash(\texttt{type}\ t_i;\chi)\ \text{decl}}$$

$$\frac{A\vdash\phi\ \text{module type}\qquad A;(\texttt{module}\ z_i:\phi)\vdash\chi\ \text{decl}}{A\vdash(\texttt{module}\ z_i:\phi;\chi)\ \text{decl}}$$

$$\frac{\begin{array}{c}A\vdash\tau\ \text{type}\qquad A\vdash\chi\ \text{decl}\\ \text{Aucun nom n'est lié plusieurs fois dans }\chi\end{array}}{A\vdash(\texttt{sig}\ (\tau)\ \chi\ \texttt{end})\ \text{module type}}$$

$$\frac{\begin{array}{c}A\vdash\Phi'\ \text{module type}\\ A;(\texttt{module}\ z_i:\Phi')\vdash\Phi\ \text{module type}\end{array}}{A\vdash(\texttt{functor}\ (z_i:\Phi')\to\Phi)\ \text{module type}}$$

$$\frac{A\vdash\Phi\ \text{module type}\qquad\alpha\notin\text{FV}(A)}{A\vdash\forall(\alpha)\ \Phi\ \text{module type}}$$

FIG. 6.6: Bonne formation des types

$$\text{(Variable)}\quad \frac{A(z) = (\texttt{module}\ z_i : \Phi)}{A \vdash z : \Phi/z_i}$$

$$\text{(Projection)}\quad \frac{A \vdash \tilde{p} : (\texttt{sig}\ (\tau)\ \chi\ \texttt{end}) \qquad (\texttt{module}\ z_i : \Phi) \in \chi \qquad A + \chi \vdash \Phi \approx \Phi'}{A \vdash \tilde{p}.z : \Phi'}$$

$$\text{(Structure)}\quad \frac{A \vdash b : \phi}{A \vdash (\texttt{struct}\ b\ \texttt{end}) : \phi}$$

$$\text{(Contrainte)}\quad \frac{A \vdash m : \Phi' \qquad A \vdash \tilde{\Phi} : \Phi \qquad A \vdash \Phi' <: \Phi}{A \vdash (m : \tilde{\Phi}) : \Phi}$$

$$\text{(Foncteur)}\quad \frac{A \vdash \tilde{\Phi}' : \Phi' \qquad A; (\texttt{module}\ z_i : \Phi') \vdash m : \Phi}{A \vdash (\texttt{functor}\ (z : \tilde{\Phi}') \to m) : (\texttt{functor}\ (z_i : \Phi') \to \Phi)}$$

$$\text{(App)}\quad \frac{\begin{array}{c} A \vdash m : (\texttt{functor}\ (z_i : \Phi'_1) \to \Phi_1) \qquad A \vdash m' : \Phi'_2 \\ A \vdash \Phi'_2 <: \Phi'_1 \qquad A; (\texttt{module}\ z_i : \Phi'_2) \vdash \Phi_1 \approx \Phi_2 \end{array}}{A \vdash (m\ m') : \Phi_2}$$

$$\text{(Sig-Vide)}\quad \frac{}{A \vdash \epsilon : (\texttt{sig}\ (\tau)\ \epsilon\ \texttt{end})}$$

$$\text{(Sig-Comp)}\quad \frac{A \vdash d : \psi \qquad A; \psi \vdash b : \texttt{sig}\ (\tau)\ \chi\ \texttt{end}}{A \vdash (d; b) : (\texttt{sig}\ (\tau)\ \psi; \chi\ \texttt{end})}$$

$$\text{(Valeur)}\quad \frac{A \vdash e : T}{A \vdash (\texttt{val}\ x = e) : (\texttt{val}\ x_i : T)}$$

$$\text{(Module)}\quad \frac{A \vdash m : \Phi}{A \vdash (\texttt{module}\ z = m) : (\texttt{module}\ z_i : \Phi)}$$

$$\text{(Type)}\quad \frac{A \vdash \tilde{\tau} : \tau \qquad \mathrm{FV}(\tau) = \emptyset}{A \vdash (\texttt{type}\ t = \tilde{\tau}) : (\texttt{type}\ t_i = \tau)}$$

FIG. 6.7: Règles de typage : modules simples

typage est supposé bien formé ; les variables liées dans des types de modules sont supposées disjointes des variables liées dans l'environnement. La bonne formation des types est définie dans la figure 6.6 : tout constructeur de type apparaissant libre dans un type ou un type de module doit être lié dans l'environnement ; de plus, deux composants de la même signature ne doivent pas avoir le même nom.

La figure 6.7 donne les règles de typage du langage de module sans extension. Ce sont les règles de typage habituelles. Dans la règle VARIABLE, le type du module z est enrichi de manière à rendre apparent que ses composants de type viennent du module de nom z_i, à l'aide de l'opérateur de renforcement Φ/z_i décrit dans la figure 6.8. Dans la règle PROJECTION, le type Φ du module z_i peut dépendre d'autres composants de la structure χ. Le type de $\tilde{p}.z$ est par conséquent un type Φ' équivalent à Φ mais ne dépendant pas de ces composants. Les autres règles sont claires.

(les règles structurelles sont omises)

$$
\begin{array}{rcl}
(\texttt{functor}\ (z_i : \Phi') \to \Phi)/p & \longrightarrow & \texttt{functor}\ (z_i : \Phi') \to \Phi/(p\ z_i) \\
(\texttt{sig}\ (\tau)\ \chi\ \texttt{end})/p & \longrightarrow & \texttt{sig}\ (\tau)\ \chi/p\ \texttt{end} \\
(\texttt{module}\ z_i : \Phi)/p & \longrightarrow & \texttt{module}\ z_i : \Phi/p.z \\
(\texttt{type}\ t_i)/p & \longrightarrow & \texttt{type}\ t_i = p.t \\
(\texttt{type}\ t_i = \tau')/p & \longrightarrow & \texttt{type}\ t_i = p.t
\end{array}
$$

FIG. 6.8: Renforcement d'un type

$$
\begin{array}{ccc}
\text{(INST)} & \text{(GEN)} & \text{(QUANT)} \\
\dfrac{A \vdash m : \forall(\alpha)\ \Phi}{A \vdash m : \Phi[\tau/\alpha]} &
\dfrac{A \vdash m : \Phi \qquad \alpha \notin \mathrm{FV}(A)}{A \vdash m : \forall(\alpha)\ \Phi} &
\dfrac{A;\ (\texttt{var}\ \alpha_i = \tau) \vdash m : \Phi}{A \vdash ([\alpha]\ m) : \Phi}
\end{array}
$$

FIG. 6.9: Quantification des types de modules

La figure 6.9 rappelle les règles d'instanciation et de généralisation des types de modules présentées précédemment en section 6.3. Les règles INST et GEN sont la transposition directe aux types de modules des règles d'instanciation et de généralisation des schémas de types. La règle QUANT introduit la variable de type syntaxique α dans l'environnement et empêche ainsi la généralisation des variables liées à cette variable lors du typage de l'expression de module m.

Les règles de typages des composants de classes sont donnés par la figure 6.10. La règle INHERIT est essentiellement du sucre syntaxique : c'est la règle THEN-2 qui accomplie la fusion des deux signatures de modules. Une méthode est typée comme une fonction (règle METHOD) ; le type de l'argument de cette fonction

$$
\begin{array}{cc}
\text{(THEN-2)} & \text{(INHERIT)} \\
\dfrac{A \vdash d : \phi \qquad A + \phi \vdash b : \phi'}{A \vdash (d;\ b) : (\phi +_A \phi')} &
\dfrac{A \vdash m : \phi}{A \vdash (\texttt{inherit}\ m) : \phi}
\end{array}
$$

$$
\text{(METHOD)}\quad
\dfrac{A \vdash (\texttt{fun}\ x \to e) : \tau' \to \tau \qquad A \vdash \tau' \approx \langle\, y : \tau;\ \omega \,\rangle}{A \vdash (\texttt{method}\ y = \varsigma(x)\ e) : (\texttt{sig}\ (\tau')\ \texttt{method}\ y\ \texttt{end})}
$$

FIG. 6.10: Règles de typage des méthodes et de l'héritage

est le type τ, et le type de son résultat est le type de la méthode y dans τ.

La règle de typage de l'opérateur `new` est semblable à celle d'Objective ML. Le type de l'objet `new` m est le type τ, et toutes les méthodes de l'objet doivent être définies (prémisse de droite).

$$\text{(New)}\quad \frac{A \vdash m : \big(\mathtt{sig}\ (\tau)\ \chi\ \mathtt{end}\big) \qquad A \vdash \tau \approx \langle\, y : \tau_y \,\rangle^{(\mathtt{method}\ y) \in \chi}}{A \vdash \big(\mathtt{new}\ m\big) : \tau}$$

6.4.3 Traduction de Objective ML

Le langage présenté précédemment contient tous les éléments syntaxique d'Objective ML, excepté les abstractions de classes et les applications de classes. Mais ces derniers éléments peuvent être considérés comme du sucre syntaxique et définis par traduction d'Objective ML dans le langage que nous venons de présenter. La traduction $[\![e]\!]$ d'une expression e d'Objective ML est définie par induction sur la syntaxe : les règles pour l'abstraction et l'application de classe sont celles données ci-dessous. Les autres règles sont évidentes.

$$\begin{array}{l} [\![\mathtt{fun}\ x \to c]\!] = \\ \quad [\alpha]\ [\alpha']\ \big(\mathtt{functor}\ (z : \mathtt{sig}\ (\alpha')\ \mathtt{val}\ x : \alpha\ \mathtt{end}) \to \\ \qquad\qquad [\![c]\!][z.x/x]\big) \\ [\![c\ e]\!] = [\![c]\!]\ \big(\mathtt{struct}\ \mathtt{val}\ x = [\![e]\!]\ \mathtt{end}\big) \end{array}$$

6.5 Impact de l'extension

Cette extension est conservative. Elle préserve en particulier l'expressivité du système de modules. Nous conjecturons sa correction. En effet, l'ajout de la quantification du type des modules par des variables de type ne devrait modifier que localement une preuve de correction, sans introduire de difficulté. Nous n'avons pas entrepris de preuve car la sémantique d'un système de modules est compliquée et l'importance de l'extension que nous proposons ne nous paraît pas suffisante pour justifier une longue preuve de correction.

Avec cette extension, il est possible de rendre implicite la paramétrisation des foncteurs par des types : il n'est plus nécessaire de définir un type abstrait dans la signature de l'argument d'un foncteur pour paramétriser un module par un type. Voici par exemple le type d'un foncteur définissant des ensembles :

```
functor
  (ord : sig type u; val compare : u → u → int end) →
  sig
     type t; val empty : t; val add : t → ord.u → t; ...
  end
```

Lors de l'application de ce foncteur, aussi bien le type u des éléments de l'ensemble que la fonction de comparaison *compare* doivent être fournis. Par contre, avec notre extension, le type u n'est plus nécessaire et le foncteur peut avoir le type suivant :

$$
\begin{array}{l}
\forall(\alpha)\ \big(\mathtt{functor} \\
\qquad (ord : \mathtt{sig}\ \mathtt{val}\ compare : \alpha \to \alpha \to int\ \mathtt{end}) \to \\
\qquad \mathtt{sig} \\
\qquad\quad \mathtt{type}\ \alpha\ t;\ \mathtt{val}\ empty : \alpha\ t; \\
\qquad\quad \mathtt{val}\ add : \alpha\ t \to \alpha \to \alpha\ t;\ \ldots \\
\qquad \mathtt{end}\big)
\end{array}
$$

Dans le type de ce foncteur, le constructeur de type t a un paramètre de type qui est contraint à être une variable explicitement quantifiée α. Il est en effet nécessaire de rendre explicite le fait que le type t dépende de la variable α : supposons par exemple que les ensembles ont une fonction *max* retournant le plus grand élément d'un ensemble ; dans le dernier type de module, le type $\forall(\alpha)\ t \to \alpha$ ne serait pas correct, alors que le type $\forall(\alpha)\ \alpha\ t \to \alpha$ l'est.

Lors de l'application de ce foncteur, seule la fonction de comparaison doit être fournie ; le type des éléments de l'ensemble est inféré. Ce type peut même être polymorphe, ce qui ne serait pas possible sans extension. En effet, si on applique un foncteur du type précédent à une structure de type

$$\mathtt{sig}\ \mathtt{val}\ compare : \forall(\alpha)\ \alpha \to \alpha \to int\ \mathtt{end}$$

on obtient un module de type

$$
\begin{array}{l}
\forall(\alpha)\ \mathtt{sig} \\
\qquad \mathtt{type}\ \alpha\ t;\ \mathtt{val}\ empty : \alpha\ t; \\
\qquad \mathtt{val}\ add : \alpha\ t \to \alpha \to \alpha\ t;\ \ldots \\
\quad \mathtt{end}
\end{array}
$$

Ce type est moralement équivalent à

$$
\begin{array}{l}
\mathtt{sig} \\
\quad \mathtt{type}\ \alpha\ t;\ \mathtt{val}\ empty : \forall(\alpha)\ \alpha\ t; \\
\quad \mathtt{val}\ add : \forall(\alpha)\ \alpha\ t \to \alpha \to \alpha\ t;\ \ldots \\
\mathtt{end}
\end{array}
$$

(dans le sens que l'on peut passer d'une valeur ayant l'un de ces types à une valeur ayant l'autre de ces types par l'application d'un foncteur qui est essentiellement l'identité).

6.6 Typage des modules

Cette extension ne complique pas significativement le typage des modules. Le principal changement est la généralisation et l'instantiation du type des modules. De plus, afin d'éviter qu'un constructeur de type puisse être unifié avec une variable de type liée en dehors de la portée de ce constructeur, un algorithme d'unification sous préfixe doit être utilisé. Un tel algorithme est facile à mettre en œuvre, et est en fait déjà utilisé dans Objective Caml, qui permet de définir des composants de modules dont le type n'est pas clos.

6.7 Fonctionnalités impératives

Afin d'assurer la correction du typage, le polymorphisme de valeurs [32] devrait *a priori* être appliqué également pour les expressions de modules, c'est-à-dire que seules les expressions de modules qui sont syntaxiquement des valeurs (chemins et foncteurs, mais ni structures ni applications de foncteurs) devraient être généralisées[1]. En effet, l'évaluation de ces expressions ne peut pas produire d'effet de bord. Cette restriction résulte malheureusement dans notre cas en une perte importante d'expressivité. Ainsi, l'exemple des ensembles polymorphes de la section 6.5 page 148 n'est plus valide. Cela n'est pas un problème pour les classes cependant, car les classes sont habituellement des foncteurs.

Une manière de contourner ce problème est d'enrichir les types des modules, afin de distinguer les foncteurs qui s'évaluent sans effet de bord des autres foncteurs. En effet, il est correct de généraliser le type du résultat de l'application d'un foncteur s'il ne produit pas d'effet de bord.

6.8 Limitations

Bien que notre proposition permette de combiner classes et modules en une seule construction, le problème de l'existence de deux styles de programmation subsiste. Par exemple, on peut fournir un type abstrait et des fonctions manipulant des valeurs de ce type :

$\texttt{module}\ \mathit{stack} : \texttt{sig type}\ \alpha\ t;\ \ldots\ \texttt{end} = \texttt{struct}$
  $\texttt{type}\ \alpha\ t = \alpha\ \mathit{list};$
  $\texttt{val}\ \mathit{empty} = [\,];$
  $\texttt{val}\ \mathit{push}\ l\ x = x :: l;$
  $\texttt{val}\ \mathit{pop}\ (x :: l) = (x, l);$
  $\ldots$
$\texttt{end}$

ou bien définir des méthodes fournissant les mêmes fonctionnalités :

$\texttt{module}\ \mathit{stack} = [\alpha]\ \texttt{struct}$
  $\texttt{val}\ \mathit{impl} = ([\,] : \alpha\ \mathit{list});$
  $\texttt{method}\ \mathit{push} =$
    $\varsigma(y)\ \texttt{fun}\ x \rightarrow \{\langle \mathit{impl} = x :: \mathit{impl}\rangle\};$
  $\texttt{method}\ \mathit{pop} =$
    $\varsigma(y)\ \texttt{let}\ (x :: l) = \mathit{impl}\ \texttt{in}\ (x, \{\langle \mathit{impl} = l\rangle\});$
  $\ldots$
$\texttt{end}$

Dans ce cas très particulier, il est en fait possible de définir un unique module

[1] La règle de sous-typage des modules correspondant à la généralisation devrait également être modifiée.

compatible avec les deux styles de programmation :

```
module stack = [α'] struct
  type α t = α list;
  val empty = [];
  val push x l = x :: l;
  val pop (x :: l) = (x, l);
  ...;
  val impl = (empty : α' t);
  method push =
    ς(y) fun x → {⟨impl = push impl x⟩};
  method pop =
    ς(y) let (x, l) = pop impl in (x, {⟨impl = l⟩});
  ...
end
```

Mais pour chaque fonction, une méthode correspondante doit être écrite, qui ne fait qu'appeler la fonction.

Il serait intéressant de pouvoir écrire n'importe quel module de manière à ce qu'il puisse être utilisé aussi bien comme une classe que comme un vrai module, sans avoir à écrire de code redondant. Comme premier pas en direction de ce but, on peut remarquer qu'il serait possible de produire à partir de la classe *stack* un module fournissant les fonctionnalités du premier module *stack* :

```
module stack' : sig type α t; ... end = struct
  type α t = α stack;
  val empty = new stack;
  val push s x = s#push x;
  val pop s = s#pop;
  ...
end
```

Une telle construction peut être facilement généralisée à toutes les définitions de classe : un module peut être associé à toute classe ; ce module exporte le type des objets de la classe (dans l'exemple ci-dessus, α *stack*), une fonction correspondant au constructeur d'objets (*empty*) et une fonction pour chaque méthode (*push*, *pop*, ...) qui appelle cette méthode.

En pratique, cependant, on serait plutôt intéressé par une construction permettant de passer d'un module à une classe. Il est en effet courant d'écrire une classe fournissant une interface à un type de donnée abstrait. Par exemple, considérons l'implémentation des ensembles par des arbres équilibrés. Il paraît naturel de définir les arbres équilibrés à l'aide d'un type de données abstrait et d'utiliser une classe comme interface. Mais il n'est pas du tout évident de produire systématiquement une classe à partir d'un module. En effet, un module peut définir plusieurs types de données (par exemple, pour les ensembles, il définit le type des éléments et le type des ensembles), mais seulement l'un de ces types doit être choisi comme type de l'état des objets de la classe. Un autre problème est que l'un des arguments des fonctions doit être singularisé, et il n'est pas toujours évident de savoir lequel, même si l'on peut souvent se baser sur leurs types. Ainsi, une fonction telle que la différence entre deux ensembles prend deux arguments du même type. Le choix entre l'un ou l'autre de ces arguments est alors complètement arbitraire. Enfin, cette fonction nécessite de pouvoir accéder au contenu d'un autre objet (qui devrait être privé).

6.9 Travaux apparentés

Reppy et Riecke avaient déjà proposé de coder des classes dans des modules [29]. Leur encodage utilise Object ML, une extension de SML avec des objets simples et du sous-typage, mais sans classe. Il ne nécessite pas d'autre extension. Il manque par conséquent de concision. Par exemple, la fonction de création doit être explicitement définie pour chaque classe ; cette fonction crée un nouvel objet et remplit explicitement ses composants. Du sucre syntaxique devrait être ajouté afin de simplifier le codage, mais la manière dont cela pourrait être réalisé n'est pas du tout claire. D'un autre côté, il devrait être possible d'adapter notre proposition à Object ML. Comme les classes d'Object ML ne sont pas polymorphes, la quantification par des types des modules ne serait pas nécessaire.

La quantification par des types des expressions de modules a été également suggérée par [16]. Sa proposition diffère de la nôtre en divers points. Nous illustrons cette différence sur un exemple :

$$
\begin{array}{l}
\texttt{struct} \\
\quad \texttt{module}\ z_1 = \texttt{functor}\ (z' : \texttt{sig type}\ t;\ \ldots\ \texttt{end}) \rightarrow \\
\qquad\qquad\qquad \texttt{struct}\ \ldots\ \texttt{end}; \\
\quad \texttt{typevar}\ \alpha; \\
\quad \texttt{type}\ t_1 = \alpha; \\
\quad \texttt{module}\ z_2 = z_1\ \big(\texttt{struct type}\ t = t_1;\ \ldots\ \texttt{end}\big) \\
\texttt{end}
\end{array}
$$

Dans [16], les variables de types sont explicitement introduites, à l'aide d'une nouvelle construction `typevar`. Une définition de type dans la portée d'une variable de type dépend implicitement de cette variable (qui peut donc apparaître dans la partie droite de la définition de type, comme c'est le cas ici pour le type t_1). Par conséquent, lorsqu'il est utilisé en dehors de la portée de la variable, le constructeur de type ainsi défini prend un paramètre supplémentaire.

Cette proposition semble plus ou moins équivalente à la nôtre. Ainsi, l'exemple précédent peut être adapté comme suit (le type t est rendu implicite, et le type t_1 est paramétrisé explicitement) :

$$
\begin{array}{l}
\texttt{struct} \\
\quad \texttt{module}\ z_1 = [\alpha]\ \texttt{functor}\ (z' : \texttt{sig}\ \ldots\ \texttt{end}) \rightarrow \\
\qquad\qquad\qquad\quad \texttt{struct}\ \ldots\ \texttt{end}; \\
\quad \texttt{type}\ \alpha\ t_1 = \alpha; \\
\quad \texttt{module}\ z_2 = z_1\ \big(\texttt{struct}\ \ldots\ \texttt{end}\big) \\
\texttt{end}
\end{array}
$$

Cependant, sa proposition semble plus difficile à combiner avec des fonctionnalités impératives. Supposons par exemple que dans le définition du module z_2 ci-dessus la structure contienne une définition d'exception `exception` E `of` t. Le type t de l'exception ne doit pas être polymorphe. Mais comme les paramètres de types peuvent être implicites, cela ne peut pas être vérifié localement, et doit apparaitre d'une manière ou d'une autre dans la signature du module

z_1. En effet, dans la définition de ce module, le type t est alors lié au type t_1 dont l'expansion est la variable de type α. Par conséquent, cette variable de type ne devrait pas être généralisable. Au contraire, avec notre proposition, la définition de l'exception serait `exception` E `of` α, et il suffirait d'interdire la généralisation de la variable de type α.

Conclusion

La contribution majeur de ce travail est bien évidemment le développement de la partie objets d'Objective Caml. Ce langage est actuellement le seul langage fonctionnel possédant des objets à avoir dépassé le stade de prototype.

Sur un plan plus théorique, nous avons montré que la synthèse de type était possible dans un langage à objets. Nous pensons également que ce travail apporte une meilleur compréhension du typage des objets. En effet, il démontre notamment que un typage très expressif des objets peut être obtenu tout en restant dans la cadre de polymorphisme à la ML. Les variables de rangées sont la clé de ce résultat, bien qu'elles n'aient été introduites à l'origine qu'afin de permettre la synthèse de type.

Notre travail sur le masquage *a posteriori* des vues, bien qu'il n'ait pas pu être utilisé pour Objective Caml, ouvre cependant un vaste espace de recherche. Il montre qu'une meilleur modularité des classes n'est pas incompatible avec un système de type riche. Par ailleurs, la notion de vues pourrait être réutiliser dans d'autres situations où l'on a besoin de considérer un objet différemment suivant le contexte.

Il apparaît [4] que l'une des meilleurs façon de représenter des structures algébriques, comme des groupes ou des polynômes dans Objective Caml est d'utiliser simultanément des classes (pour leur extensibilité) et des modules (pour l'abstraction). Par ailleurs, de nombreux travaux (notamment [10]) ont lieu actuellement sur les *mixins*, des modules extensibles. Il serait intéressant de reconsidérer notre proposition de fusion des classes et des modules au vu de ces résultats.

Bibliographie

[1] Martín Abadi and Luca Cardelli. A theory of primitive objects: Second-order systems. *Science of Computer Programming*, 25(2-3):81–116, December 1995. Preliminary version appeared in D. Sanella, editor, Proceedings of European Symposium on Programming, pages 1-24. Springer-Verlag, April 1994.

[2] Martín Abadi and Luca Cardelli. *A theory of objects*. Springer, 1996.

[3] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *Transactions on Programming Languages and Systems*, 15(4):575–631, September 1993.

[4] Sylvain Boulmé, Thérèse Hardin, and Renaud Rioboo. Modules, objects et calcul formel. In *Actes des Journées francophones des langages applicatifs (JFLA'99)*, pages 171–188, February 1999.

[5] M. Brandt and F. Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticae*, 33:309–338, apr 1998.

[6] Kim B. Bruce, Adrian Fiech, and Leaf Petersen. Subtyping is not a good "match" for object-oriented languages. In *ECOOP*, number 1241 in LNCS, pages 104–127. Springer-Verlag, 1997.

[7] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John Mitchell. F-bounded quantification for object-oriented programming. In *Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, September 1989.

[8] Luca Cardelli and Peter Wegner. On understanding types, data abstraction and polymorphism. *Computing surveys*, 17(4):471–522, December 1985.

[9] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *9th symposium Principles of Programming Languages*, pages 207–212. ACM Press, 1982.

[10] D.Ancona and E. Zucca. A theory of mixin modules: Algebraic laws and reduction semantics. Technical Report DISI-TR-99-05, 1999. Submitted for journal publication.

[11] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *Object-Oriented Programming Systems, Languages, and Applications*, 1995.

[12] J. Eifrig, S. Smith, and V. Trifonov. Type inference for recursively constrained types and its application to OOP. In *Mathematical Foundations of Programming Semantics*, 1995.

[13] K. Fisher and J.C. Mitchell. A delegation-based object calculus with subtyping. In *Fundamentals of Computation Theory (FCT'95)*, number 965 in LNCS, pages 42–61. Springer-Verlag, 1995.

[14] Kathleen Fisher and John H. Reppy. The design of a class mechanism for Moby. In ACM-SIGPLAN, editor, *Conference on Programming Language Design and Implementation*. ACM Press, 1999.

[15] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *21st symposium Principles of Programming Languages*, pages 123–137. ACM Press, 1994.

[16] Stefan Kahrs. First-class polymorphism for ML. Technical report, University of Edinburgh, 1993.

[17] Claude Kirchner and Jean-Pierre Jouannaud. Solving equations in abstract algebras: a rule-based survey of unification. Research Report 561, Université de Paris Sud, Orsay, France, April 1990.

[18] Xavier Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 6(5):667–698, 1996.

[19] Xavier Leroy, Jérôme Vouillon, Damien Doligez, et al. The Objective Caml system. Software and documentation available on the Web, `http://pauillac.inria.fr/ocaml/`, 1996.

[20] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Information and Computation*, 100(1):1–40, September 1992.

[21] Benjamin C. Pierce and David N. Turner. Statically typed friendly functions via partially abstract types. Technical Report ECS-LFCS-93-256, University of Edinburgh, LFCS, April 1993. Also available as INRIA-Rocquencourt Rapport de Recherche No. 1899.

[22] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994. A preliminary version appeared in Principles of Programming Languages, 1993, and as University of Edinburgh technical report ECS-LFCS-92-225, under the title "Object-Oriented Programming Without Recursive Types".

[23] François Pottier. *Synthèse de types en présence de sous-typage: de la théorie à la pratique*. PhD thesis, Université Paris VII, July 1998.

[24] Didier Rémy. Syntactic theories and the algebra of record terms. Research Report 1869, Institut National de Recherche en Informatique et Automatisme, BP 105, F-78 153 Le Chesnay Cedex, 1993.

[25] Didier Rémy. Programming objects with ML-ART: An extension to ML with abstract and record types. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 321–346. Springer-Verlag, April 1994.

[26] Didier Rémy. *Des enregistrements aux objets*. Mémoire d'habilitation à diriger des recherches, Université de Paris 7, 1998.

[27] Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory And Practice of Object Systems*, 4(1):27–50, 1998. A preliminary version appeared in the proceedings of the 24th ACM Conference on Principles of Programming Languages, 1997.

[28] Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory And Practice of Objects Systems*, 1998. Available at `ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/objective-ml!tapos98.ps.gz`.

[29] John H. Reppy and Jon G. Riecke. Classes in Object ML. Presented at the FOOL'3 workshop, July 1996.

[30] Jon G. Riecke and Christopher A. Stone. Privacy via subsumption. *Theory and Practice of Object Systems*, 1999.

[31] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[32] Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–356, December 1995.

[33] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1), November 1994.