# Unions de variants polymorphes abstraits

Romain Bardou

1er septembre 2006

## 1 Introduction

Ce rapport de stage montre mon travail effectué à l'université de Nagoya, Japon, avec Jacques Garrigue. C'est une destination très sympathique pour faire un stage de cinq mois. L'université propose de nombreux clubs pour vous intégrer, ainsi que des cours de japonais très bien faits et très réguliers si vous voulez apprendre la langue. De plus, Nagoya se trouve au centre du Japon, entre la capitale, Tokyo, et l'ex-capitale, Kyoto. Le seul bémol concerne le climat, très chaud et humide en été...

Pendant mon stage, j'ai travaillé sur les unions de variants polymorphes abstraits. Programmer des projets de grande taille demande une certaine organisation. Découper le programme en parties indépendantes permet de réduire les risques d'erreurs et de déléguer une partie du travail. Ce découpage est facilité si le langage de programmation fournit des outils de haut niveau tels que les objets ou les modules. Les variants sont un outil qui peut participer à la modularité d'un programme ; en particulier les variants polymorphes abstraits, aussi appelés variants polymorphes privés.

Dans une première partie, nous décrirons informellement ce que sont les variants polymorphes privés. Ensuite, nous décrirons la fonctionnalité que l'on va leur ajouter : les unions, en montrant quel est leur intérêt. Les parties 4, 5 et 6 développent le problème de façon technique. On étend les variants polymorphes abstraits et en on établit un modèle ; ensuite, on montre comment vérifier la validité d'une union de variants privés, par rapport au modèle.

## 2 Variants polymorphes abstraits

### 2.1 Variants

Les *variants* sont un type de base des langages fonctionnels tels que OCaml. Ils permettent de décrire des structures de données composées de plusieurs *cas*. Par exemple, si on veut une structure de données qui permet de contenir tantôt des entiers, et tantôt des booléens, on peut utiliser le type suivant :

```
type t = Entier of int | Booleen of bool
```

Une fois ce type `t` défini, on peut créer des valeurs de ce type :

```
let zero = Entier 0
let vrai = Booleen true
```

On peut aussi utiliser le mécanisme du *pattern-matching* pour, étant donnée une valeur de type `t`, déterminer s'il s'agit d'un `Entier` ou d'un `Booleen` et récupérer la valeur associée :

```
let f x = match x with
    Entier e -> e
  | Booleen b -> if b then 1 else 0
```

Cette fonction `f` renvoie un entier qui dépend de l'*étiquette* de `x` (`Entier` ou `Booleen`) et de la valeur associée à cette étiquette. Appliquons cette fonction à différentes valeurs de type `t` :

```
# f zero;;
- : int = 0
# f vrai;;
- : int = 1
# f (Booleen false);;
- : int = 0
```

## 2.2 Variants polymorphes

Les *variants polymorphes* sont une extension des variants. Contrairement à ces derniers, pour créer une valeur d'un type variant polymorphe, on n'a pas besoin de déclarer ce type (et la liste des étiquettes qui le composent). Réécrivons les exemples précédents en utilisant des variants polymorphes :

```
# let zero = `Entier 0
  let vrai = `Booleen true;;
val zero : [> `Entier of int ] = `Entier 0
val vrai : [> `Booleen of bool ] = `Booleen true
```

Comme on le voit, on a utilisé les étiquettes `` `Entier `` et `` `Booleen `` sans les définir. Le symbole `` ` `` permet de savoir qu'il s'agit d'étiquettes de variants polymorphes.

Le compilateur infère automatiquement le type associé à ces valeurs. Par exemple, le type associé à `zero` est `[> `Entier of int]`. Ceci décrit un type variant polymorphe qui contient au moins l'étiquette `` `Entier ``, associée au type `int`. Ceci est différent du type `[`Entier of int]` qui décrit un variant polymorphe qui ne contient que l'étiquette `` `Entier ``, et aucune autre.

En instanciant le type `[> `Entier of int]`, on peut en déduire que `zero` a aussi les types :

- `[`Entier of int]` (uniquement l'étiquette `` `Entier ``),
- `[`Entier of int | `Booleen of bool]` (deux étiquettes : `` `Entier `` et `` `Booleen ``),
- `[> `Entier of int | `Booleen of bool]` (au moins les étiquettes `` `Entier `` et `` `Boolean ``),
- ...

Le typeur infère aussi le type de fonctions qui utilisent des variants polymorphes :

```
# let f x = match x with
      `Entier e -> e
    | `Booleen b -> if b then 1 else 0;;
val f : [< `Booleen of bool | `Entier of int ] -> int = <fun>
```

Ici, le type `[< `Booleen of bool | `Entier of int ]` signifie : n'importe quel type qui possède au plus les étiquettes `` `Booleen `` (associée au type `bool`) et `` `Entier `` (associée au type `int`). Ceci signifie en particulier que l'on peut appliquer `f` à `zero` et à `vrai` :

```
# f zero;;
- : int = 0
# f vrai;;
```

```
- : int = 1
# f (`Booleen false);;
- : int = 0
```

En effet, `zero` a le type `[> `Entier of int]`, donc peut être vu comme une valeur de type `[`Entier of int]`, qui est une instance du type `[< `Booleen of bool | `Entier of int ]`.

## 2.3 Variants privés

### 2.3.1 Abstraction

Le mécanisme d'*abstraction* est très important pour la modularité des programmes. Un programme peut être découpé en *modules*. Un module est souvent représenté par un fichier dans les langages de programmation. En Caml, il existe un langage de module, qui permet d'avoir plusieurs modules dans un même fichier, et qui étend les possibilités de ces modules, en particulier avec les notions d'abstraction et de *foncteurs*.

Prenons un module décrivant un langage très simple, composé des entiers et des booléens, en utilisant les variants (non polymorphes) :

```
module M = struct
  type t = Entier of int | Booleen of bool
  let f = function
      Entier e -> e
    | Booleen b -> if b then 1 else 0
end
```

Comme on le voit dans cet exemple, un module peut être une *structure* (`struct`) composée de types (ici, un seul type : le type `t`) et de valeurs (ici, une seule valeur : la fonction `f`). Ce module possède une signature, qui représente son type et qui décrit son contenu :

```
module M : sig
  type t = Entier of int | Booleen of bool
  val f : t -> int
end
```

Comme on le voit, l'implémentation de la fonction `f` n'apparait pas dans la signature. Ceci est déjà une forme d'abstraction : l'implémentation de `f` est oubliée, il ne reste que son type. Ce qui signifie qu'on peut utiliser `f`, mais qu'on ne peut pas faire de supposition concernant son implémentation. L'avantage est que l'on peut modifier l'implémentation du module (pour étendre, corriger ou modifier le comportement de `f`) sans avoir à modifier le reste du code source.

Mais l'implémentation du type `t`, elle, n'a pas été oubliée. Or, il peut être intéressant d'oublier l'implémentation de ce type si l'on veut pouvoir la changer plus tard. Ceci peut se faire en imposant une signature à un module :

```
module M' : sig
  type t
  val f : t -> int
end = M
```

L'implémentation du type `t` a été masquée, mais on sait toujours qu'il existe, ce qui est important puisque `t` apparait dans le type de `f`. Par contre, puisqu'on ne connait plus l'implémentation de `t`, on ne peut plus créer de valeurs du type `t`. Seule l'implémentation du module pourra le faire, ce qui assure que si

l'on désire changer le type `t`, on n'aura pas besoin de modifier autre chose que l'implémentation de `M'`.

Le mécanisme d'abstraction permet donc au programmeur de forcer une utilisation particulière de ses modules afin de renforcer leur indépendance.

### 2.3.2 Variants polymorphes et abstraction

Dans l'exemple de la section précédente, on oublie entièrement le type `t`. Les variants polymorphes possèdent un mécanisme d'abstraction particulier, utilisant le mot-clé `private`, permettant de n'oublier qu'une partie du type. Une partie ou toutes ses étiquettes sont oubliées, mais on sait toujours qu'il s'agit d'un variant polymorphe. Par exemple, le module suivant est abstrait pour masquer l'étiquette `'Entier` de son type `t` :

```
module M'' : sig
  type t = private [> Booleen of bool]
  val f : t -> int
end = struct ... end
```

On sait qu'*il existe* une implémentation pour `t`, et que cette implémentation possède l'étiquette `Booleen` associée au type `bool`. Mais elle peut posséder d'autres étiquettes.

# 3 Unions de variants polymorphes

## 3.1 Variants concrets

OCaml permet, dans les définitions de types variants polymorphes, d'utiliser le nom d'un autre type variant polymorphe concret pour éviter d'avoir à recopier ses étiquettes.

```
type t = ['Entier of int | 'Booleen of bool]
type u = [t | 'Chaine of string]
```

La définition de `u` est automatiquement, directement expansée :

```
type u = ['Entier of int | 'Booleen of bool | 'Chaine of string]
```

Il n'y a donc pas de modification fondamentale du système de type. Une fois les définitions expansées, on se retrouve avec des variants polymorphes classiques.

## 3.2 Variants abstraits

Le mécanisme d'expansion ne permet pas d'écrire une définition telle que :

```
type t = private [> ]
type u = [t | 'Chaine of string]
```

En effet, `t` étant abstrait, on ne connait pas l'ensemble de ses étiquettes et de leur type associé. On ne peut donc pas faire l'expansion de `t` dans la définition de `u`.

OCaml ne permet donc pas de telles définitions. Nous allons montrer comment on peut autoriser ces définitions en étendant le système de type. Cette extension consistera à conserver les noms des types abstraits dans les définitions de types, sans les expanser.

### 3.3 Compatibilité

Un problème se pose immédiatement lorsque l'on essaye d'écrire la définition suivante :

```
type t = private [> ]
type u = [t | `Chaine of string]
```

`t` est abstrait (partiellement : on sait que c'est un variant polymorphe), donc on ne connait pas son implémentation, qui pourrait être n'importe quoi. Que se passe-t-il si l'implémentation de `t` utilise l'étiquette `` `Chaine ``?

```
type t = [`Chaine of int]
```

Le type `u` serait donc :

```
type u = [`Chaine of int | `Chaine of bool]
```

Le type `u` associe l'étiquette `` `Chaine `` à deux types différents : `int` et `bool`. Ce conflit ne peut pas être détecté sans connaître l'implémentation de `t`.

Pour résoudre ce problème, on propose d'étendre les définitions de types variants polymorphes privées, en ajoutant des informations de *compatibilité*. Par exemple, on peut modifier la définition de `t` en ajoutant une compatibilité avec `` `Chaine of string `` :

```
type t = private [> ]~[`Chaine of string]
```

Cette définition signifie que si `t` contient l'étiquette `` `Chaine ``, alors cette étiquette a pour type associé `string`. Cette compatibilité devra être vérifiée au moment où l'on abstrait `t`.

On pourra aussi ajouter une compatibilité avec l'étiquette `` `Chaine ``, quel que soit son type associé :

```
type t = private [> ]~[`Chaine of _]
```

Ceci signifie en fait que `t` ne contient pas l'étiquette `` `Chaine ``.

Pour finir, imaginons que nous voulions faire l'union de deux variants privés :

```
type t = private [> ]
type u = private [> ]
type v = [t | u]
```

Le type `v` est l'union du type `t` et du type `u`. Mais cette union n'est possible que s'il n'y a pas de conflits. Imaginons par exemple l'implémentation suivante :

```
type t = [`A of int]
type u = [`A of bool]
```

Avec cette implémentation, la définition du type `v` entraine un conflit sur l'étiquette `` `A ``. Il faut donc rajouter une information de compatibilité :

```
type t = private [> ]
type u = private [> ]~[t]
type v = [t | u]
```

Ici, on affirme que `u` est compatible avec `t`, ce qui permet de faire leur union. Cette compatibilité entre `u` et `t` est vérifiée au moment de leur abstraction, pour vérifier qu'il n'y a effectivement pas de conflit.

En résumé, les informations de compatibilité nous donne certains renseignements sur la partie abstraite d'un variant, dans le but d'unir un variant privé avec d'autres étiquettes ou variants.

## 3.4 Pattern-matching sur les unions

Dans OCaml, on peut utiliser, dans un pattern-matching, la syntaxe `#t` pour désigner toutes les étiquettes du type `t`.

```
type t = [`A of int | `B of bool]

let is_t = function
    #t -> true
  | _ -> false
```

La fonction `t` renvoie true si, et seulement si, son paramètre est de type `t`. En réalité, le compilateur expanse simplement la définition de `t` :

```
let is_t = function
    (`A _ | `B _) -> true
  | _ -> false
```

Le type de `is_t` est `[> t ] -> bool`.

On ne peut faire ceci qu'avec des types concrets, puisqu'on ne peut pas expanser les types abstraits (dont on ne connait pas l'ensemble des étiquettes). Pour utiliser le pattern-matching sur des unions de types abstraits et pouvoir différencier les cas, on propose d'étendre cette fonctionnalité à tous les variants polymorphes.

## 3.5 Motivation

Les unions de variants polymorphes abstraits peuvent être utiles pour modulariser un programme. Imaginons par exemple que l'on veuille construire un langage à partir de plusieurs sous-langages. Chaque sous-langage sera un module, et on utilisera un foncteur (une fonction des modules dans les modules) pour construire l'union de ces sous-langages.

Voici par exemple deux sous-langages :

```
module Entier = struct
  type t = [`Entier of int]
  let f = function `Entier e -> e
end
module Booleen = struct
  type t = [`Booleen of bool]
  let f = function `Booleen b -> if b then 1 else 0
end
```

Chaque module défini un type `t`, et une fonction `f`. Remarquez qu'il s'agit de la même fonction `f` que dans les exemples des sections précédentes, mais qu'on a séparé chaque cas dans un module correspondant.

On veut maintenant réunir ces deux sous-langages, pour obtenir un type `t` contenant à la fois les `` `Entier `` et les `` `Booleen ``, et une fonction `f` réunissant les cas définis dans chaque sous-langage. Pour cela, on utilise un foncteur `Union`, prenant en paramètres deux sous-langages `A` et `B` et les réunissant en un seul :

```
module Union
  (A: sig
    type t = private [> ]
    val f: t -> int
  end)
  (B: sig
    type t = private [> ]~[A.t]
    val f: t -> int
```

```
  end) =
struct
  type t = [A.t | B.t]
  let f = function
      #A.t as x -> A.f x
    | #B.t as x -> B.f x
end
```

Ce foncteur effectue l'union des types privés `A.t` et `B.t`. Il est nécessaire qu'ils soient compatibles ; on ajoute donc une information de compatibilité dans la définition de `B.t`. On peut ensuite construire `t`, l'union de `A.t` et `B.t`. On construit aussi `f`, l'union de `A.f` et `B.f`, en utilisant le symbole `#` qui permet, dans un pattern-matching, de traiter tous les cas d'un type donné. Ici, on l'utilise avec des types abstraits.

Maintenant qu'on a ce foncteur, on peut faire l'union de n'importe quels modules ayant chacun un type variant polymorphe `t` compatibles, et une fonction `f` de type `t -> int`. Par exemple, on peut faire l'union des sous-langages `Entier` et `Booleen` :

```
module EntierBooleen = Union(Entier)(Booleen)
```

En prenant en compte la possibilité d'effectuer des récursions ouvertes, on peut obtenir une solution au problème de l'expression.

# 4 Modèle

On va d'abord modéliser les définitions de variants privés. Ces définitions sont une extension des définitions actuelles, afin d'ajouter des informations de compatibilité. Le modèle permettra de caractériser un type à l'aide de ses étiquettes et de leur type associé.

## 4.1 Définitions de types

La syntaxe des définitions de type est la suivante :

$$
\begin{array}{rcl}
\delta & ::= & \texttt{type}\ t = \mathcal{D} \\
\\
\mathcal{D} & ::= & [A] \\
 & | & \texttt{private}\ [> A]\ \tilde{}\ [C] \\
\\
A & ::= & (l\ \texttt{of}\ \tau | t)* \\
\\
C & ::= & (?l\ \texttt{of}\ \tau | \neg l | ?t)*
\end{array}
$$

Ici, $l$ représente les étiquettes, $\tau$ les types et $t$ les noms de types. L'ensemble des étiquettes sera noté $\mathcal{L}$, l'ensemble des types $T$ et l'ensemble des noms de types $N$.

Les éléments de la forme $l$ `of` $\tau$ ou $t$ seront appelés *associations* et seront souvent notés $A$.

Les éléments de la forme $?l$ `of` $\tau$, $\neg l$ ou $?t$ seront appelés *compatibilités* et seront souvent notés $C$.

On notera souvent $R$ un élément qui est soit une association, soit une compatibilité. Les définitions de type elles-mêmes seront souvent notées $\delta$.

On note $\mathbb{A}[\delta]$ l'ensembles des associations de $\delta$, c'est-à-dire l'ensemble des éléments à gauche du symbole ~. On note $\mathbb{C}[\delta]$ l'ensembles des compatibilités de $\delta$, c'est-à-dire l'ensemble des éléments à droite du symbole ~. Si $\delta$ est fermée, $\mathbb{C}[\delta]$ est vide.

Une définition de type est donc soit concrète (et donc *fermée*) et composée uniquement d'associations, soit privée (*ouverte*). Si la définition est ouverte, elle est définie par sa borne inférieure, composée d'associations, et par ses compatibilités. Ses compatibilités sont de plusieurs formes :

- $?l$ `of` $\tau$ indique que le type défini est compatible avec l'étiquette $l$, à condition qu'elle ait le type associé $\tau$,
- $\neg l$ indique que l'étiquette $l$ n'apparait pas dans l'implémentation du type défini et donc que ce type est compatible avec cette étiquette $l$, quelque soit son type associé,
- $?t$ indique que le type défini est compatible avec le type $t$.

**Remarque 1** *La syntaxe ci-dessus ne fait pas intervenir les types variants privés bornés, de la forme :*

```
type t = private [< U_1 | ... | U_n > L_1 | ... | L_m]
```

*En effet, on traitera ces types comme des types concrets, de la forme :*

```
type t = [U_1 | ... | U_n]
```

*En effet, pour notre problème, on n'a besoin que des informations sur la borne supérieure, puisqu'elle contient déjà toutes les étiquettes qui peuvent être utilisées.*

*Cependant, ce choix a un impact sur le modèle et sur les unions que l'on pourra effectuer. En particulier, ce type d'union sera refusé :*

```
type t = private [< `A of int | `B of int]
type u = [t | `A of bool]
```

*Si* `t` *était implémenté par* `[`B of int]`*, le type* `u` *pourrait avoir un sens. Mais jamais le type* `t` *ne pourra posséder l'étiquette* `` `A ``*. Or, cette contrainte apparait après que le type* `t` *ait été défini. Une telle modification* a posteriori *n'est pas souhaitable en particulier dans un cadre modulaire. De plus, il n'est pas clair, en lisant cette définition, si le type associé à* `` `A `` *d'un élément ayant le type* `u` *est* `` `int `` *ou* `` `bool ``*, ce qui peut géner lors d'un pattern-matching sur un tel élément.*

**Remarque 2** *Une définition fermée ne possède pas de compatibilités. En effet, les informations de compatibilité nous renseignent sur la partie abstraite d'un type, plus particulièrement sur les étiquettes qu'il pourrait y avoir. Or, un type fermé n'ajoute aucune abstraction. Il peut être défini à partir de types abstraits, mais il ne peut pas ajouter d'informations de compatibilité sur ces types qui ne soient pas déjà connues.*

## 4.2 Unions de fonctions

### 4.2.1 Définition

Pour définir le modèle, on aura besoin de la notion d'unions de fonctions.

Toute fonction $f : A \to B$ peut être vue comme une relation, donc un élément de $\mathcal{P}(A \times B)$. On peut donc définir l'union de deux fonctions $f$ et $g$ comme l'union sur l'ensemble des relations. Cette union sera notée $f \cup g$.

De même, on note $f \subseteq g$ l'inclusion de fonctions, correspondant à l'inclusion des relations associées.

Si une union de fonctions est une fonction, c'est-à-dire si chaque élément $a \in A$ a pour image un singleton, alors on dit que cette union est *cohérente* et on note cette fonction $f \dot{\cup} g : A \to B$. Si l'union est incohérente, on notera $f \dot{\cup} g = \bot$.

### 4.2.2 Fonction partielle et $\varnothing$

Les fonctions partielles seront explicites. Si $f$ n'est pas définie en $a$, on notera $f(a) = \varnothing$. $\varnothing$ doit appartenir à l'ensemble d'arrivée. On notera $B_\varnothing = B \cup \{\varnothing\}$.

Si $f(a) = \varnothing$, alors $(f \cup g)(a) = \{g(a)\}$. Ceci permet à une union d'être cohérente en $a$ si l'une des deux fonctions n'est pas définie en $a$.

## 4.3 Modèle des définitions de types

### 4.3.1 Valuations

**Definition 1 (Valuation)**
*Une* valuation *est un élément de* $N \to \mathcal{L} \to T_\varnothing$. *L'ensemble des valuations sera noté* $\mathcal{V}$.

Une valuation associe, à chaque type, une *interprétation*. Chaque type est caractérisé par l'ensemble de ses étiquettes et des types associés. Donc une interprétation d'un type $t$ associe, à chaque étiquette $l$, le type associé à cette étiquette $l$, ou $\varnothing$ si cette $l$ n'apparait pas dans $t$.

On définit la valuation étendue de v, notée $\widetilde{v}$, ainsi :

$$\begin{array}{rcl} \widetilde{v}(t) & = & v(t) \\ \widetilde{v}(?t) & = & v(t) \\ \widetilde{v}(l \texttt{ of } \tau) & = & \{l \mapsto \tau\} \\ \widetilde{v}(?l \texttt{ of } \tau) & = & \{l \mapsto \tau\} \\ \widetilde{v}(\neg l) & = & \{l \mapsto \Omega\} \end{array}$$

$\Omega$ est un type qui n'est jamais utilisé.

**Definition 2 (Satisfaction)**
*Une valuation* $v$ satisfait *une définition de type* $\delta$ *si :*

$$\forall A, R \in \mathbb{A}[\delta] \times (\mathbb{A}[\delta] \cup \mathbb{C}[\delta]),\ \widetilde{v}(A) \cup \widetilde{v}(R) \text{ est cohérent}$$

On va construire un modèle $W$, ensemble de valuations. Les définitions de types devront être satisfaites par toutes les valuations de $W$.

### 4.3.2 Construction du modèle

On veut modéliser un ensemble de définitions $\delta_1 \cdots \delta_n$. On procède de manière itérative ; on part d'un modèle $W_0 = \mathcal{V}$, l'ensemble des valuations. Puis, pour chaque définition $\delta_i$, on construit $W_i$ à partir de $W_{i-1}$. Cette construction est représentée par une fonction $V$ de la façon suivante :

$$W_i = V_{W_{i-1}}(\delta_i)$$

Pour définir le modèle, il suffit donc de définir $V_W(\delta)$.

### 4.3.3 Types fermés

Si $\delta$ définit le type fermé $t$, on défini $V_W(\delta)$ de la façon suivante :

$$V_W(\delta) = \{v \in W \mid v(t) = \dot{\bigcup_{A \in \mathbb{A}[\delta]}} \widetilde{v}(A)\}$$

Ainsi, on ne conserve que les valuations dont l'interprétation de $t$ correspond à sa définition.

### 4.3.4 Types ouverts

Si $\delta$ définit le type ouvert $t$, on commence par définir $V^0_W(\delta)$ :

$$V^0_W(\delta) = \{v \in W \mid v(t) \supseteq \dot{\bigcup_{A \in \mathbb{A}[\delta]}} \widetilde{v}(A)\}$$

Ainsi, on élimine les valuations dont l'interprétation ne contient pas la borne inférieure de $t$.

Ensuite, on peut définir $V_W(\delta)$ à partir de $V^0_W(\delta)$ :

$$V_W(\delta) = \{v \in V^0_W(\delta) \mid \forall C \in \mathbb{C}[\delta],\ v(t) \cup \widetilde{v}(C) \text{ est cohérent}\}$$

On ne conserve donc que les valuations qui respectent les compatibilités de $t$.

## 4.4 Validité dans le modèle

### 4.4.1 Définition

**Definition 3 (Validité vis-à-vis de $W$)**
*Une définition $\delta$ est* valide *vis-à-vis du modèle $W$ si :*
– *pour tout $v \in W$, $v$ satisfait $\delta$*
– *et si pour tout $t$ ou $?t$ apparaissant dans $\delta$, $t$ a déjà été défini.*

Autrement dit, une définition $\delta$ sera invalide si l'un des cas suivant se présente :
– deux de ses associations sont incompatibles entre elles,
– une de ses compatibilités n'est pas vérifiable,
– $\delta$ mentionne des types qui n'ont pas encore été définis.

On supposera par la suite que le modèle $W$ n'aura été construit qu'à partir de définitions valides.

### 4.4.2 Types non définis

Le fait de ne construire $W$ qu'à partir de types valides nous assure la propriété suivante :

**Propriete 1 (Validité et types non définis)**
*À une étape $W$ donnée, si $D$ est l'ensemble des types ayant déjà été définis, et $\bar{D}$ l'ensemble des types n'ayant pas encore été définis, alors :*

$$\forall v \in W,\ \forall v' \in \bar{D} \to \mathcal{L} \to T_\varnothing,\ v|_D \cup v' \in W$$

Cette propriété assure que si un type n'a pas encore été défini, alors on peut lui associer l'interprétation que l'on veut. Ainsi, quand on voudra définir ce type, on sera sûr que toutes les interprétations possibles pour ce type seront disponibles.

### 4.4.3 Types déjà définis

Une autre propriété intéressante est que si un type est valide, alors l'ajouter au modèle ne change pas les interprétations disponibles pour les types ayant déjà été définis.

**Propriete 2 (Validité et types déjà définis)**
*À une étape $W$ donnée, si $D$ est l'ensemble des types ayant déjà été définis, si $\delta$ est une définition de type, alors :*

$$\delta \text{ est valide} \iff \begin{cases} \{v|_D \mid v \in W\} = \{v|_D \mid v \in V_W(\delta)\} \\ \text{et les types mentionnes par } \delta \text{ sont dans } D \end{cases}$$

Notez que l'on a ainsi une caractérisation de la validité.

# 5 Traduction des définitions de types

En pratique, les définitions de types ne seront pas utilisées telles quelles lors du test de validité. Elles seront traduites avant, et ces traductions seront placées dans un environnement.

## 5.1 Traduction

Comme pour la création du modèle, on traduit chaque définition de type une par une, dans le même ordre. Les traductions des types fermés sont placées dans un environnement $\Delta$, et les traductions des types ouverts sont placées dans un environnement $\Gamma$. Les traductions sont des ensembles d'associations et de compatibilités, et les environnements sont des fonctions des noms de types vers leur traduction. Si un type $t$ n'a pas encore été ajouté à $\Delta$, alors $\Delta(t) = \bot$, et de même pour $\Gamma$.

Le but de la traduction est d'expanser les définitions de types fermés pour que leurs noms n'apparaissent plus, que ce soit dans les associations ou dans les compatibilités. De plus, la traduction réunit les associations et les compatibilités ensembles.

Pour cela, on utilise une fonction de traduction $\mathcal{T}_\Delta$. Celle-ci utilise l'environnement des types fermés $\Delta$ pour expanser les définitions de type fermés, mais n'a pas besoin de l'environnement des types ouverts $\Gamma$. La fonction de traduction

est définie ainsi :

$$\begin{array}{rcl}
\mathcal{T}_\Delta(\delta) & = & \bigcup_{R \in \mathbb{A}[\delta] \cup \mathbb{C}[\delta]} \widetilde{\Delta}(R) \\
\widetilde{\Delta}(l \texttt{ of } \tau) & = & \{l \texttt{ of } \tau\} \\
\widetilde{\Delta}(t) & = & \begin{cases} \Delta(t) & \text{si } \Delta(t) \neq \bot \\ \{t\} & \text{sinon} \end{cases} \\
\widetilde{\Delta}(?t) & = & \{?A \mid A \in \Delta(t)\} \\
\widetilde{\Delta}(?l \texttt{ of } \tau) & = & \{?l \texttt{ of } \tau\} \\
\widetilde{\Delta}(\neg l) & = & \{\neg l\}
\end{array}$$

Remarquez que les traductions des types fermés ne contiennent pas de compatibilités.

## 5.2 Modèle

Avant d'utiliser cette traduction, on va montrer qu'elle ne perd ni n'ajoute aucune information. Pour cela, on modélise les traductions et on montre que le modèle est conservé pendant la traduction.

On va donc définir $[\![t]\!]_{\Gamma,\Delta,v}$, l'interprétation du type $t$ sous les environnements $\Gamma$ et $\Delta$, et sous la valuation $v$.

### 5.2.1 Types fermés

Si $t$ est un type fermé, alors son interprétation après traduction est définie ainsi :

$$[\![t]\!]_{\Gamma,\Delta,v} = \dot{\bigcup_{A \in \Delta(t)}} \widetilde{v}(A)$$

Cette définition rappelle la définition de $V_W(\delta)$ si $\delta$ est la définition d'un type fermé.

### 5.2.2 Types ouverts

Si $t$ est un type ouvert, on commence par définir $[\![t]\!]^0_{\Gamma,\Delta,v}$ :

$$[\![t]\!]^0_{\Gamma,\Delta,v} = v(t) \cup \dot{\bigcup_{A \in \mathbb{A}[\Gamma(t)]}} \widetilde{v}(A)$$

Cette définition rappelle la définition de $V^0_W(\delta)$ si $\delta$ est la définition d'un type ouvert. Ici $v(t)$ joue le rôle de la partie abstraite, remplaçant l'inclusion dans la définition de $V^0_W(\delta)$.

On peut ensuite définir $[\![t]\!]_{\Gamma,\Delta,v}$ à partir de $[\![t]\!]^0_{\Gamma,\Delta,v}$ :

$$[\![t]\!]_{\Gamma,\Delta,v} = \begin{cases} [\![t]\!]^0_{\Gamma,\Delta,v} & \text{si } \forall C \in \mathbb{C}[\Gamma(t)],\ [\![t]\!]^0_{\Gamma,\Delta,v} \cup \widetilde{v}(C) \text{ est cohérent} \\ \bot & \text{sinon} \end{cases}$$

Une fois encore, cette définition rappelle celle de $V_W(\delta)$ si $\delta$ est la définition d'un type ouvert.

### 5.2.3 Conservation du modèle

On peut maintenant prouver que le modèle est conservé pendant la traduction.

**Propriete 3 (Conservation du modèle)**
*A une etape* $(W, \Gamma, \Delta)$ *donnee, si* $\delta$ *définit* $t$*, on a :*

$$V_W(\delta) = \{v \in W \mid v(t) = [\![t]\!]_{\Gamma,\Delta,v}\}$$

# 6 Test de validité

La validité dans le modèle est définie à partir des valuations. On va proposer un test de validité à partir des traductions des types, et montrer qu'il est équivalent à la validité dans le modèle.

## 6.1 Définition

La validité dans le modèle consiste à vérifier, pour chaque valuation, que les associations d'une définition $\delta$ sont compatibles entre elles, et que ses compatibilités sont compatibles avec ses associations. De plus, on demande que les noms de types qui apparaissent dans $\delta$ correspondent à des types déjà définis. On va donc s'approcher de cette définition, mais sans la quantification sur les valuations.

Pour cela, on définit une relation de compatibilité $\odot$ entre deux ensembles d'associations $\mathcal{A}_1$ et $\mathcal{A}_2$, sous l'environnement $\Gamma$ :

$$\Gamma \vdash \mathcal{A}_1 \odot \mathcal{A}_2$$

L'environnement $\Delta$ n'est pas utile : on utilise des définitions traduites, sans noms de types fermés. Avec cette relation, on peut définir le test de validité :

**Definition 4 (Test de validité)**
*Une définition de type* $\delta$ *est valide vis a vis de* $\Gamma$ *et de* $\Delta$ *si et seulement si :*

$$\Gamma \vdash \mathbb{A}[\mathcal{T}_\Delta(\delta)] \odot (\mathbb{A}[\mathcal{T}_\Delta(\delta)] \cup !\mathbb{C}[\mathcal{T}_\Delta(\delta)])$$

*Avec (si* $\Omega$ *est un type qui n'est jamais utilisé) :*

$$\begin{array}{rcl} ![R_1|\cdots|R_n] & = & [!R_1|\cdots|!R_n] \\ !\neg l & = & l \texttt{ of } \Omega \\ !?l \texttt{ of } \tau & = & l \texttt{ of } \tau \\ !?t & = & t \end{array}$$

*Et si les types mentionnés par* $\delta$ *sont déjà définis.*

Les associations (à gauche du symbole $\odot$) seront testées vis-à-vis des associations (à droite du symbole $\odot$), et des compatibilités (à droite du symbole $\odot$). Ces compatibilités sont vues comme des associations. La compatibilité $\neg l$, qui signifie "compatible avec toute étiquette $l$", est traduite en $l$ `of` $\Omega$. $\Omega$ étant un type qui n'est jamais utilisé, s'il y a une incompatibilité, c'est qu'une association utilise $l$.

## 6.2 Relation de compatibilité

Il faut maintenant définir $\Gamma \vdash \mathcal{A}_1 \odot \mathcal{A}_2$ de telle façon que si cette relation est vraie, alors pour toute valuation, il y a cohérence entre chaque paire d'association de $\mathcal{A}_1 \times \mathcal{A}_2$.

On suppose que chaque $t$ correspond a un type ouvert. Ceci permet d'etre certain que si on rencontre un tel type, l'absence d'information (association ou compatibilite) n'est pas une information en soi. C'est pour cette raison qu'on doit, lors de la traduction, faire l'expansion des types fermés. Notez que ceci permet aussi de se passer de $\Delta$ pour tester la compatibilité.

On utilise aussi une relation d'*héritage*, notée $\Gamma \vdash R \Subset t$, qui est vraie si $t$ hérite de $R$, c'est-à-dire si $R$ apparait dans $t$, éventuellement à travers un autre type apparaissant dans $t$.

On utilise les règles suivantes :

$$\frac{\Gamma \vdash A_1 \odot \mathcal{A} \ \cdots \ \Gamma \vdash A_n \odot \mathcal{A}}{\Gamma \vdash \{A_1, \cdots, A_n\} \odot \mathcal{A}}\text{Disp1} \quad \frac{\Gamma \vdash A \odot A_1 \ \cdots \ \Gamma \vdash A \odot A_n}{\Gamma \vdash A \odot \{A_1, \cdots, A_n\}}\text{Disp2}$$

$$\frac{\Gamma \vdash A \Subset t}{\Gamma \vdash A \odot t}\text{In} \quad \frac{R \in \Gamma(t)}{\Gamma \vdash R \Subset t}\text{In1} \quad \frac{t' \in \Gamma(t) \quad \Gamma \vdash A \Subset t'}{\Gamma \vdash A \Subset t}\text{In2}$$

$$\frac{?t' \in \Gamma(t) \quad \Gamma \vdash A \Subset t'}{\Gamma \vdash ?A \Subset t}\text{In3} \quad \frac{\Gamma \vdash ?l \ \texttt{of} \ \tau \Subset t}{\Gamma \vdash l \ \texttt{of} \ \tau \odot t}\text{LT1} \quad \frac{\neg l \in \Gamma(t)}{\Gamma \vdash l \ \texttt{of} \ \tau \odot t}\text{LT2}$$

$$\frac{\Gamma \vdash ?l \ \texttt{of} \ \tau_1 \Subset t \quad \Gamma \vdash ?l \ \texttt{of} \ \tau_2 \Subset t \quad \tau_1 \neq \tau_2}{\Gamma \vdash l \ \texttt{of} \ \tau \odot t}\text{LT3}$$

$$\frac{l \neq l' \text{ or } \tau = \tau'}{\Gamma \vdash l \ \texttt{of} \ \tau \odot l' \ \texttt{of} \ \tau'}\text{LL} \quad \frac{\Gamma \vdash ?t \Subset t'}{\Gamma \vdash t \odot t'}\text{TT} \quad \frac{\Gamma \vdash A' \odot A}{\Gamma \vdash A \odot A'}\text{Sym}$$

Les règles $Disp1$ et $Disp2$ se contentent de tester la compatibilité de chaque paire de $\mathcal{A}_1 \times \mathcal{A}_2$. Ensuite, il y a plusieurs cas possibles : test entre deux étiquettes, entre deux noms de types ouverts ou entre une étiquette et un nom de type ouvert.

Le test entre deux étiquettes (règle $LL$) est immédiat : $l_1$ `of` $\tau_1$ et $l_2$ `of` $\tau_2$ sont incompatibles si, et seulement si, $l_1 = l_2$ et $\tau_1 \neq \tau_2$.

Le test entre deux noms de types est effectué par la règle $TT$ et la règle $In$ (ces règles étant éventuellement précédées par une règle $Sym$). Pour prouver que $t$ est compatible avec $t'$, soit on prouve que $t$ apparait explicitement dans $t'$ (règles $In$ et $In1$) ou par héritage (règles $In$ et $In2$), soit on prouve que $?t$ apparait explicitement dans $t'$ (règles $TT$ et $In1$) ou par héritage (règles $In$ et $In3$).

C'est le même principe avec la règle $LT1$ et la règle $In$, qui s'occupent du cas où l'on veut tester la compatibilité d'une étiquette avec un nom de type ouvert. Mais on a aussi besoin de deux règles supplémentaires : $LT2$ et $LT3$. En effet, si on sait qu'une étiquette $l$ n'apparait pas dans un type $t$, alors $t$ est compatible avec $l$, quelque soit le type associé à l'étiquette. Et il y a deux moyens de savoir que $l$ n'apparait pas dans $t$ : soit $\neg l$ apparait en tant que compatibilité (règle $LT2$), soit $l$ apparait deux fois avec deux types différents dans les compatibilités de $t$ (règle $LT3$). En effet, si $t$ possède l'étiquette $l$, alors le type associé à $l$ doit être $\tau_1$ et $\tau_2$ à la fois, ce qui n'est pas possible.

**Remarque 3** *La règle LT3 est nécessaire pour être complet par rapport au modèle. Mais elle pose un problème pratique ; en effet, il est nécessaire de pouvoir prouver l'inégalité de deux types. Il s'agit d'une nouvelle notion non traitée dans les systèmes usuels. Donc dans une implémentation de ces règles, on peut vouloir se passer de la complétude en n'implémentant pas LT3.*

*Si un test de compatibilité échoue à cause de l'absence de cette règle, c'est que soit le programmeur a mis explicitement une compatibilité* $?l$ `of` $\tau_1$ *et une compatibilité* $?l$ `of` $\tau_2$ *avec* $\tau_1 \neq \tau_2$*, et donc il peut les remplacer par* $\neg l$ *pour faire réussir le test ; soit ces deux compatibilités sont héritées, et alors il n'est pas sûr que cette compatibilité était attendue par le programmeur.*

**Propriete 4**
*Il existe un algorithme qui, étant donnés* $\Gamma$, $\mathcal{A}_1$ *et* $\mathcal{A}_2$*, décide* $\Gamma \vdash \mathcal{A}_1 \odot \mathcal{A}_2$.

## 6.3 Correction et complétude

On commence par prouver deux lemmes, le premier allant servir pour montrer la correction du test de validité, le deuxième allant servir pour montrer la complétude.

**Lemme 1 (Correction)**
*Si* $t$ *a déjà été défini à l'étape* $(W, \Gamma, \Delta)$*, alors pour tout* $v \in W$ *:*

1. $\Gamma \vdash ?l$ `of` $\tau \Subset t \Longrightarrow \widetilde{v}(t) \cup \widetilde{v}(?l$ `of` $\tau)$ *est cohérent*
2. $\Gamma \vdash ?t' \Subset t \Longrightarrow \widetilde{v}(t) \cup \widetilde{v}(?t')$ *est cohérent*
3. $\neg l \in \Gamma(t) \Longrightarrow \widetilde{v}(t) \cup \widetilde{v}(\neg l)$ *est cohérent*
4. $\Gamma \vdash l$ `of` $\tau \Subset t \Longrightarrow \widetilde{v}(l$ `of` $\tau) \subseteq \widetilde{v}(t)$
5. $\Gamma \vdash t' \Subset t \Longrightarrow \widetilde{v}(t') \subseteq \widetilde{v}(t)$
6. $\Gamma \vdash ?l$ `of` $\tau_1 \Subset t$ *et* $\Gamma \vdash ?l$ `of` $\tau_2 \Subset t$ *et* $\tau_1 \neq \tau_2 \Longrightarrow \widetilde{v}(t) \cup \widetilde{v}(\neg l)$ *est cohérent*

Ce premier lemme décrit quel est l'effet sur le modèle de chaque élément, association ou compatibilité, qui apparait dans l'environnement (éventuellement par héritage). Par exemple, si on sait qu'une étiquette apparait comme une association dans $t$, alors cette étiquette est incluse dans toute interprétation de $t$ (point 4).

**Lemme 2 (Complétude)**
*Si* $t$ *a déjà été défini à l'étape* $(W, \Gamma, \Delta)$ *et est ouvert, alors pour tout* $t'$ *ouvert, et pour tout* $(l, \tau_0)$ *:*

1. $\left.\begin{array}{r} \Gamma \not\vdash ?t' \Subset t \\ \Gamma \not\vdash t' \Subset t \end{array}\right\} \Longrightarrow \exists v \in W,\ \widetilde{v}(t) \cup \widetilde{v}(t')$ *soit incohérent*

2. $\left.\begin{array}{r} \Gamma \not\vdash ?l \text{ of } \tau_0 \Subset t \\ \Gamma \not\vdash l \text{ of } \tau_0 \Subset t \\ \neg l \notin \Gamma(t) \\ Card\{\tau \mid \Gamma \not\vdash ?l \text{ of } \tau \Subset t\} \leq 1 \end{array}\right\} \Longrightarrow \exists v \in W,\ \widetilde{v}(t)(l) \notin \{\Omega, \tau_0, \varnothing\}$

Ce lemme montre que si on ne peut pas prouver qu'un type $t$ est cohérent avec une association $A$ donnée, alors il existe une valuation qui rend incohérent l'union de $t$ et de $A$. Le premier point concerne le cas où $A$ est un nom de type tandis que le second point concerne le cas où $A$ est une étiquette.

A l'aide de ces deux lemmes, on peut prouver la correction et la complétude du test de validité :

**Propriete 5 (Correction et complétude)**
*A une étape $(W, \Gamma)$ donnée, pour toute définition de type $\delta$, la validité de $\delta$ vis à vis de $W$ est équivalente à la validité de $\delta$ vis a vis de $\Gamma$ et de $\Delta$.*

La preuve de cette propriété consiste principalement à relier le mécanisme de *dispatch* de $\odot$ et la quantification sur $A$ et $R$ dans la définition de la validité dans le modèle, à l'aide des deux lemmes précédents.

# 7 Conclusion

On a défini un test de validité visant à assurer la cohérence de définitions de variants polymorphes contenant des noms de types privés. Pour cela, on a étendu les définitions de variants avec des informations de compatibilités, utilisées par une relation de compatibilité $\odot$ après une traduction $\mathcal{T}_\Delta$.

On a modélisé les définitions de types et prouvé la correction et la complétude du test de validité vis-à-vis de ce modèle.

Il reste cependant à étendre le système de type lui-même :

- prendre en compte les noms de types qui apparaissent dans les définitions de types variants polymorphes (qui ne peuvent plus être directement expansées),
- utiliser la relation de compatibilité à bon escient.

Et il faut modifier la compilation des modules contenant des définitions de types privés, ainsi que le pattern-matching sur les types contenant des unions de variants privés, afin de pouvoir utiliser le symbole `#` sur ceux-ci.

# A Preuves

On note, si $A$ est un ensemble de fonctions de $B \to C$ et si $B' \subseteq B$ :

$$A|_{B'} = \{f|_{B'} \mid f \in A\}$$

## A.1 Validité et types non définis

**Propriete 1 (Validité et types non définis)**
*À une étape $W$ donnée, si $D$ est l'ensemble des types ayant déjà été définis, et $\bar{D}$ l'ensemble des types n'ayant pas encore été définis, alors :*

$$\forall v \in W,\ \forall v' \in \bar{D} \to \mathcal{L} \to T_\varnothing,\ v|_D \cup v' \in W$$

**Preuve**
On prouve la propriete par recurrence sur les etapes $W_0 \cdots W_n$.

**Cas de base** $W_0 = \mathcal{V}$, donc la propriete est verifiee trivialement pour le cas de base.

**Cas general** Soit $v \in W_i$. Soit $v' \in \bar{D_i} \to \mathcal{L} \to T_\varnothing$. Soit $w = v|_{D_i} \cup v'$. Montrons que $w \in W_i$. On suppose pour l'instant que $\delta_i$ est une definition concrete.

$v|_{D_{i-1}} \in W_i|_{D_{i-1}}$ et $v' \cup v|_{t_i} \in \bar{D_{i-1}} \to \mathcal{L} \to T_\varnothing$, donc par hypothese de recurrence :

$$w = v|_{D_{i-1}} \cup (v' \cup v|_{t_i}) \in W_{i-1}$$

$v \in W_i$ implique que $v(t_i) = \dot{\bigcup\limits_{R \in \mathbb{A}[\delta_i]}} \widetilde{v}(R)$. De plus, la validite de $\delta_i$ nous assure que $\delta_i$ ne mentionne pas de type $t \in D_i$. Donc :

$$\forall R \in \mathbb{A}[\delta_i],\ \widetilde{v}(R) = \widetilde{w}(R)$$

Donc :

$$w(t_i) = v(t_i) = \dot{\bigcup_{R \in \mathbb{A}[\delta_i]}} \widetilde{v}(R) = \dot{\bigcup_{R \in \mathbb{A}[\delta_i]}} \widetilde{w}(R)$$

Donc $w \in W_i^0$. Or, $v \in W_i = V_{W_{i-1}}(\delta)$, donc :

$$\forall S \in \mathbb{C}[\delta_i],\ v(t_i) \cup \widetilde{v}(S) \text{ est cohérent}$$

Et comme $\delta$ ne mentionne pas de type de $\bar{D_i}$, cette relation est aussi valable pour $w$. Donc $w \in W_i$.

La preuve est la même dans le cas ou $\delta_i$ est une definition abstraite bornee. S'il s'agit d'une definition ouverte, la seule difference est qu'on a une inclusion :

$$w(t_i) = v(t_i) \supseteq \dot{\bigcup_{R \in \mathbb{A}[\delta_i]}} \widetilde{v}(R) = \dot{\bigcup_{R \in \mathbb{A}[\delta_i]}} \widetilde{w}(R)$$

Donc $w(t_i) \supseteq \dot{\bigcup\limits_{R \in \mathbb{A}[\delta_i]}} \widetilde{w}(R)$, et donc $w \in W_i^0$. La suite de la preuve est identique.

□

## A.2 Validité et types déjà définis

**Propriete 2 (Validité et types déjà définis)**
*À une étape $W$ donnée, si $D$ est l'ensemble des types ayant déjà été définis, si $\delta$ est une définition de type, alors :*

$$\delta \text{ est valide} \iff \begin{cases} \{v|_D \mid v \in W\} = \{v|_D \mid v \in V_W(\delta)\} \\ \text{et les types mentionnes par } \delta \text{ sont dans } D \end{cases}$$

**Preuve**
On montre d'abord le sens de gauche a droite, puis la reciproque.

**Implication** L'inclusion $V_W(\delta)|_D \subseteq W|_D$ est immediate.
Soit $v \in W$. Montrons que $v|_D \in V_W(\delta)|_D$.
D'apres la propriete 1 :

$$\forall v' \in \bar{D} \to \mathcal{L} \to T_\varnothing,\ v|_D \cup v' \in W$$

Donc en particulier si $v'(t) = \dot{\bigcup}_{R \in \mathbb{A}[\delta_i]} \widetilde{v}(R)$ ou si $v'(t) \supseteq \dot{\bigcup}_{R \in \mathbb{A}[\delta_i]} \widetilde{v}(R)$, on a :

$$v|_D \cup v' \in W$$

Remarquons que ces unions sont differentes de $\perp$, d'apres la validite de $\delta$. Notons $w = v|_D \cup v'$. D'apres la validite de $\delta$, qui assure que les types mentionnes sont deja definis, on a donc :

$$w \in V_W^0(\delta)$$

Supposons maintenant que $\delta = \texttt{type}\ t = [\mathcal{R}]\tilde{}[\mathcal{R}']$. On a donc :

$$w(t) = \dot{\bigcup}_{R \in \mathbb{A}[\delta]} \widetilde{v}(R)$$

D'apres la validite de $\delta$, on a :

$$\forall (R_1, R_2) \in \mathcal{R} \times (\mathcal{R} \cup \mathcal{R}'),\ \widetilde{v}(R_1) \cup \widetilde{v}(R_2) \text{ est cohérent}$$

Par consequent, on a, pour tout $S \in \mathcal{R}'$, $w(t) \cup \widetilde{w}(S)$ est coherent. Donc $w \in V_W(\delta)$. Et donc $w|_D = v|_D \in V_W(delta)|_D$.

Le cas ou $\delta$ est une definition abstraite bornee se prouve exactement de la meme facon. Si $\delta$ est une definition ouverte, on peut choisir un $v'$ qui correspond a la borne inferieure et utiliser la meme preuve pour montrer que $v|_D \cup v' \in V_W(\delta)$, et donc que $v|_D \in V_W(\delta)|_D$.

**Reciproque** Supposons que $\delta$ (definissant $t$) ne soit pas valide. Il existe donc $R_1 \in \mathcal{R}$, $R_2 \in \mathcal{R} \cup \mathcal{R}'$ et $v \in W$ tels que $\widetilde{v}(R_1) \cup \widetilde{v}(R_2)$ soit incoherent.

Supposons d'abord que $R_2 \in \mathcal{R}$. Alors $\dot{\bigcup}_{R \in \mathcal{R}} \widetilde{v}(R) = \perp$.

Supposons qu'il existe $v' \in \bar{D} \to \mathcal{L} \to T_\varnothing$ tel que $w = v' \cup v|_D \in V_W^0(\delta)$. Alors, comme $\delta$ ne mentionne que des types deja definis, on a :

$$\dot{\bigcup}_{R \in \mathbb{A}[\delta]} \widetilde{w}(R) = \perp$$

et donc on a $v'(t) = \bot$ ou $v'(t) \supseteq \bot$, ce qui est absurde. Donc $v|_D \notin V^0_W(\delta)|_D$.

Supposons maintenant que $R_2 \in \mathcal{R}'$. Alors $v(t) \cup \widetilde{v}(R_2)$ n'est pas coherent, et donc $v|_D \notin V_W(\delta)|_D$. □

## A.3 La traduction conserve le modèle

**Propriete 3 (Conservation du modèle)**

*A une etape* $(W, \Gamma, \Delta)$ *donnee, si* $\delta$ *définit* $t$*, on a :*

$$V_W(\delta) = \{v \in W \mid v(t) = [\![t]\!]_{\Gamma,\Delta,v}\}$$

**Preuve**

On raisonne par récurrence forte sur le nombre de définitions de type. S'il n'y en a aucune, il n'y a rien à montrer. On se place donc à une étape $(W, \Gamma, \Delta)$ donnée, avec une définition $\delta$ définissant un type $t$.

Supposons d'abord que $\delta$ est une définition fermée : $\delta =$ type t $= [\mathcal{R}]$ avec $\mathcal{R} = R_1|\cdots|R_n$. On a donc, par définition du modèle :

$$\begin{aligned} [\![t]\!]_{\Gamma,\Delta,v} &= \dot{\bigcup_{R\in\Delta(t)}} \widetilde{v}(R) \\ V_W(\delta) &= \{v \in W \mid v(t) = \dot{\bigcup_{R\in\mathcal{R}}} \widetilde{v}(R)\} \end{aligned}$$

Il suffit donc de montrer que pour tout $v \in W$ :

$$\dot{\bigcup_{R\in\Delta(t)}} \widetilde{v}(R) = \dot{\bigcup_{R\in\mathcal{R}}} \widetilde{v}(R)$$

sachant que :

$$\Delta(t) = \bigcup_{i=1}^{n} \widetilde{\Delta}(R_i)$$

On raisonne pour cela par récurrence sur $n$, le nombre d'éléments de $\mathcal{R}$. Le cas de base est immédiat ($\mathcal{R} = \emptyset = \Delta(t)$). Pour montrer l'égalité au rang $i$ il suffit de montrer que :

$$\dot{\bigcup_{R\in\widetilde{\Delta}(R_i)}} \widetilde{v}(R) \overset{(1)}{=} \widetilde{v}(R_i)$$

Si $R_i = l$ `of` $\tau$, alors $\widetilde{\Delta}(R_i) = \{l$ `of` $\tau\}$. Donc l'égalité (1) est vérifiée. Si $R_i = t'$ et $t$ est ouvert, alors $\widetilde{\Delta}(R_i) = \{t'\}$ et l'égalité (1) est aussi vérifiée. Enfin, si $R_i = t'$ et $t'$ est fermé, alors $\widetilde{\Delta}(R_i) = \Delta(t')$ et l'égalité (1) devient :

$$\dot{\bigcup_{R\in\Delta(t')}} \widetilde{v}(R) = v(t')$$

Cette égalité est vérifiée, il s'agit de l'hypothèse de récurrence à l'étape où $t'$ a été défini, et $t'$ a été défini avant $t$ (sinon $t$ ne pourrait pas le mentionner).

Montrons maintenant le cas où $\delta$ est une définition ouverte :

$$\delta = \text{type t } = \text{ private } [> \mathcal{R}]\tilde{}[\mathcal{R}']$$

On peut réutiliser la preuve précédente (avec $\Gamma$ au lieu de $\Delta$) pour montrer que :

$$\dot{\bigcup_{R\in\Gamma(t)}} \widetilde{v}(R) = \dot{\bigcup_{R\in\mathcal{R}}} \widetilde{v}(R)$$

On a alors :

$$[\![t]\!]^0_{\Gamma,\Delta,v} = \dot{\bigcup_{R\in\Gamma(t)}} \widetilde{v}(R) \cup v(t) = \dot{\bigcup_{R\in\mathcal{R}}} \widetilde{v}(R) \cup v(t)$$

On a donc :

$$v(t) = [\![t]\!]^0_{\Gamma,\Delta,v} \Longrightarrow v(t) \supseteq \dot{\bigcup_{R\in\mathcal{R}}} \widetilde{v}(R)$$

De plus, si $v(t) \supseteq \dot{\bigcup_{R\in\mathcal{R}}} \widetilde{v}(R)$ alors :

$$v(t) = \bigcup_{R\in\mathcal{R}} \widetilde{v}(R) \cup v(t) = [\![t]\!]^0_{\Gamma,\Delta,v}$$

Donc :

$$v(t) = [\![t]\!]^0_{\Gamma,\Delta,v} \iff v(t) \supseteq \dot{\bigcup_{R\in\mathcal{R}}} \widetilde{v}(R)$$

Et donc :

$$V^0_W(\delta) = \{v \in W \mid v(t) = [\![t]\!]^0_{\Gamma,\Delta,v}\}$$

Montrons maintenant l'égalité de la propriété. Par definition du modele, on a :

$$\begin{array}{rcl} [\![t]\!]_{\Gamma,\Delta,v} & = & \begin{cases} [\![t]\!]^0_{\Gamma,\Delta,v} & \text{si } \forall R \in \mathbb{C}[\Gamma(t)],\ [\![t]\!]^0_{\Gamma,\Delta,v} \cup \widetilde{v}(R) \text{ est cohérent} \\ \bot & \text{sinon} \end{cases} \\ V_W(\delta) & = & \{v \in V^0_W(\delta) \mid \forall S \in \mathbb{C}[\delta],\ v(t) \cup \widetilde{v}(S) \text{ est cohérent}\} \end{array}$$

On a montré que pour tout $v \in V^0_W(\delta)$, $v(t) = [\![t]\!]^0_{\Gamma,\Delta,v}$. Donc il suffit de montrer, pour chaque élément $R_i$ de $\mathbb{C}[\delta] = \{R_1, \cdots, R_n\}$, que

$$\begin{array}{c} v(t) \cup \widetilde{v}(R_i) \text{ est cohérent} \\ \Longleftrightarrow \\ \forall R \in \widetilde{\Delta}(R_i),\ v(t) \cup \widetilde{v}(R) \text{ est cohérent} \end{array}$$

Soit $R_i \in \mathbb{C}[\delta]$. Si $R_i$ n'est pas une compatibilité $?u$ avec $u$ fermé, alors $\widetilde{\Delta}(R_n) = \{R_n\}$ et l'équivalence est immédiate. Supposons donc que $R_i = ?u$ avec $u$ fermé. On a $\widetilde{\Delta}(R_n) = ?\Delta(u)$. $u$ ayant été défini avant $t$, on peut appliquer l'hypothèse de récurrence sur sa définition $\delta_u$ et en déduire que :

$$v(u) = [\![u]\!]_{\Gamma,\Delta,v} = \bigcup_{R\in\Delta(u)} \widetilde{v}(R)$$

Soit $R \in \Delta(u)$. D'après l'égalité précédente, si $v(t) \cup \widetilde{v}(?R)$ est incohérent, alors $v(t) \cup v(u)$ est incohérent, car $\widetilde{v}(?R) = \widetilde{v}(R)$ et $\widetilde{v}(R) \subseteq v(u)$. Et si $v(t) \cup v(u)$ est incohérent, alors il existe $R \in \Delta(u)$ tel que $v(t) \cup \widetilde{v}(R)$ est incohérent, donc tel que $v(t) \cup \widetilde{v}(?R)$ est incohérent. Ceci montre l'équivalence voulue.

Donc on a bien :

$$V_W(\delta) = \{v \in W \mid v(t) = [\![t]\!]_{\Gamma,\Delta,v}\}$$

□

## A.4 Lemme de correction

**Lemme 1 (Correction)**

*Si $t$ a déjà été défini à l'étape $(W, \Gamma, \Delta)$, alors pour tout $v \in W$ :*

1. $\Gamma \vdash ?l \texttt{ of } \tau \Subset t \Longrightarrow \widetilde{v}(t) \cup \widetilde{v}(?l \texttt{ of } \tau)$ *est cohérent*
2. $\Gamma \vdash ?t' \Subset t \Longrightarrow \widetilde{v}(t) \cup \widetilde{v}(?t')$ *est cohérent*
3. $\neg l \in \Gamma(t) \Longrightarrow \widetilde{v}(t) \cup \widetilde{v}(\neg l)$ *est cohérent*
4. $\Gamma \vdash l \texttt{ of } \tau \Subset t \Longrightarrow \widetilde{v}(l \texttt{ of } \tau) \subseteq \widetilde{v}(t)$
5. $\Gamma \vdash t' \Subset t \Longrightarrow \widetilde{v}(t') \subseteq \widetilde{v}(t)$
6. $\Gamma \vdash ?l \texttt{ of } \tau_1 \Subset t$ *et* $\Gamma \vdash ?l \texttt{ of } \tau_2 \Subset t$ *et* $\tau_1 \neq \tau_2 \Longrightarrow \widetilde{v}(t) \cup \widetilde{v}(\neg l)$ *est cohérent*

**Preuve**

On suppose que $t$ est ouvert (sinon on a $\Gamma(t) = \bot$ et donc le lemme est immédiat). Soit $W_t$ le modèle juste avant de définir $t$, $\delta_t$ la définition de $t$, et $W'_t$ le modèle juste après la définition ($W'_t = V_{W_t}(\delta_t)$). On a donc, d'après la propriété 3 :

$$W'_t = \{v \in W_t \mid v(t) = [\![t]\!]_{\Gamma,\Delta,v}\}$$

Soit $v \in W$. $W \subseteq W'_t$, donc $v \in W'_t$. Donc :

$$\widetilde{v}(t) = v(t) = [\![t]\!]_{\Gamma,\Delta,v}$$

Soit $R$ une compatibilité telle que $\Gamma \vdash R \Subset t$. Raisonnons par récurrence sur l'ordre de définition des types. Si $R \in \mathbb{C}[\Gamma(t)]$, d'après la définition de $[\![t]\!]_{\Gamma,\Delta,v}$, on a :

$$[\![t]\!]_{\Gamma,\Delta,v} \cup \widetilde{v}(R) \text{ est cohérent}$$

Si $\Gamma \vdash R \Subset u$ et $u \in \Gamma(t)$, alors $u$ a été défini avant $t$ et donc par hypothèse de récurrence :

$$\widetilde{v}(R) \subseteq \widetilde{v}(u)$$

De plus, par définition de $[\![t]\!]_{\Gamma,\Delta,v}$ :

$$[\![t]\!]_{\Gamma,\Delta,v} \cup \widetilde{v}(u) \text{ est cohérent}$$

Donc :

$$[\![t]\!]_{\Gamma,\Delta,v} \cup \widetilde{v}(R) \text{ est cohérent}$$

Ce qui montre les implications 1 et 2.

Soit $\neg l \in \Gamma(t)$. Donc $\neg l \in \mathbb{C}[\Gamma(t)]$ et d'après la définition de $[\![t]\!]_{\Gamma,\Delta,v}$, on a :

$$[\![t]\!]_{\Gamma,\Delta,v} \cup \widetilde{v}(\neg l) \text{ est cohérent}$$

Ce qui montre l'implication 3.

Soit $R$ une association telle que $\Gamma \vdash R \Subset t$. Raisonnons par récurrence sur l'ordre de définition des types. Si $R \in \mathbb{A}[\Gamma(t)]$, d'après la définition de $[\![t]\!]^0_{\Gamma,\Delta,v}$, on a :

$$\widetilde{v}(R) \subseteq [\![t]\!]^0_{\Gamma,\Delta,v}$$

Si $\Gamma \vdash R \Subset u$ et $u \in \Gamma(t)$, alors $u$ a été défini avant $t$ et donc par hypothèse de récurrence et par définition de $[\![t]\!]^0_{\Gamma,\Delta,v}$ :

$$\widetilde{v}(R) \subseteq \widetilde{v}(u) \subseteq [\![t]\!]^0_{\Gamma,\Delta,v}$$

Ce qui montre les implications 4 et 5.

Soit $\tau_1$ et $\tau_2$ tels que :

$$\Gamma \vdash ?l \texttt{ of } \tau_1 \Subset t$$
$$\Gamma \vdash ?l \texttt{ of } \tau_2 \Subset t$$
$$\tau_1 \neq \tau_2$$

D'après le point 1, que l'on a déjà montré, on a donc :

$$\widetilde{v}(t) \cup \widetilde{v}(?l \texttt{ of } \tau_1) \text{ est cohérent}$$
$$\widetilde{v}(t) \cup \widetilde{v}(?l \texttt{ of } \tau_2) \text{ est cohérent}$$

Supposons que $\widetilde{v}(t)(l) \neq \varnothing$. Alors soit $\widetilde{v}(t)(l) = \tau_1$, et donc $\widetilde{v}(t)(l) \cup \widetilde{v}(?l \texttt{ of } \tau_2)$ est incohérent, ce qui est absurde, soit $\widetilde{v}(t)(l) \neq \tau_1$ et donc $\widetilde{v}(t)(l) \cup \widetilde{v}(?l \texttt{ of } \tau_1)$ est incohérent, ce qui est absurde. Donc $\widetilde{v}(t)(l) = \varnothing$, ce qui montre l'implication 6. □

## A.5 Lemme de complétude

**Lemme 2 (Complétude)**

*Si $t$ a déjà été défini à l'étape $(W, \Gamma, \Delta)$ et est ouvert, alors pour tout $t'$ ouvert, et pour tout $(l, \tau_0)$ :*

1. $\left.\begin{array}{c} \Gamma \not\vdash ?t' \Subset t \\ \Gamma \not\vdash t' \Subset t \end{array}\right\} \Longrightarrow \exists v \in W,\ \widetilde{v}(t) \cup \widetilde{v}(t')$ *soit incohérent*

2. $$\left.\begin{array}{r} \Gamma \not\vdash ?l \texttt{ of } \tau_0 \Subset t \\ \Gamma \not\vdash l \texttt{ of } \tau_0 \Subset t \\ \neg l \notin \Gamma(t) \\ Card\{\tau \mid \Gamma \not\vdash ?l \texttt{ of } \tau \Subset t\} \leq 1 \end{array}\right\} \Longrightarrow \exists v \in W,\ \widetilde{v}(t)(l) \notin \{\Omega, \tau_0, \varnothing\}$$

**Preuve**

Commençons par montrer la première implication. Soit $(v_0, l_0) \in W \times \mathcal{L}$ tel que pour tout $t \in N$, $v_0(t)(l_0) = \varnothing$. $v_0$ existe ; en effet, il y a un nombre fini de types définis $D$, chacun ayant des définitions $\delta$ finies, et on peut prendre $\mathcal{L}$ infini. Donc pour ces types on peut trouver une étiquette non utilisée $l_0$. Et pour les types non définis, on peut choisir n'importe quelle interprétation.

Soit $\tau$ et $\tau'$ deux types différents de $T$. Soit maintenant $v_1$ la valuation :

$$v_1(u)(l) = \begin{cases} \tau & \text{si } (u,l) = (t, l_0) \\ \tau' & \text{si } (u,l) = (t', l_0) \\ \tau' & \text{si } (u,l) = (t'', l_0) \text{ et } \Gamma \vdash t' \Subset t'' \\ v_0(u)(l) & \text{sinon} \end{cases}$$

Par définition, $v_1(t)(l_0) \neq v_1(t')(l_0)$ donc $\widetilde{v}_1(t) \cup \widetilde{v}_1(t')$ est incohérent. Soit $(W_0, \Gamma_0, \Delta_0)$ l'étape précédant la définition $\delta_t$ de $t$, et $W_t = V_{W_0}(\delta_t)$.

Par rapport à $v_0$, $v_1$ ne modifie que les types déjà définis (à l'étape $W_0$) suivant : $t'$ et les types $t''$ qui héritent de $t'$ ($\Gamma_0 \vdash t' \Subset t''$). $t'$ est un type ouvert ; donc on peut modifier son étiquette $l_0$. Ceci ne crée pas d'incohérence puisqu'aucun type n'était défini en $l_0$ pour $v_0$. $v_1$ respecte donc les définitions des types déjà définis, y compris ceux qui héritent de $t'$. Donc $v_1 \in W_0$.

De plus, $v_0$ était dans $W$, donc dans $W_t$. Donc il vérifiait la propriété suivante :

$$v_0(t) \supseteq \dot{\bigcup_{R \in \mathbb{A}[\delta_t]}} \widetilde{v}_0(R)$$

Cette propriété est toujours vérifiée pour $v_1$ ; en effet, on n'a fait qu'ajouter une définition pour l'étiquette $l_0$, donc la borne inférieure, est toujours incluse dans $v_1(t)$. Cette borne inférieure ne change pas, sauf si $t$ hérite de $t'$ ($\Gamma \vdash t' \Subset t$), ce qui est contradictoire avec les hypothèses. Donc $v_1 \in V^0_{W_0}(\delta_t)$.

$v_0$ vérifiait aussi la propriété suivante :

$$\forall R \in \mathbb{C}[\Gamma(t)],\ [\![t]\!]^0_{\Gamma,\Delta,v_0} \cup \widetilde{v}_0(R) \text{ est cohérent}$$

Donc s'il existe $R$ tel que $[\![t]\!]^0_{\Gamma,\Delta,v_1} \cup \widetilde{v}_1(R)$ est incohérent, cette incohérence porte sur l'étiquette $l_0$. Et donc $\widetilde{v}_1(R)(l_0)$ n'est ni $\varnothing$, ni $\tau$ (qui est le type associé à l'étiquette $t_0$ pour le type $t$). On peut supposer que $R \neq ?l_0$ `of` $\tau'$, quitte à choisir une autre étiquette $l_0$. Or, les seuls types qui associent un type différent de $\tau$ à $l_0$, sont $t'$ et les types $t''$ qui héritent de $t'$. Donc soit $R = ?t'$ et donc $?t' \in \Gamma(t)$, soit $R = ?t''$ et $\Gamma \vdash t' \Subset t''$ et donc $\Gamma \vdash ?t' \Subset t$, ce qui est absurde. Donc $v_1 \in W_t$.

On conclut en construisant $v_2 \in W$ à partir de la propriété 2 appliquée à $v_1$ plusieurs fois, de $W_t$ à $W$.

Montrons maintenant la deuxième implication. Soit $(l, \tau_0) \in \mathcal{L} \times T$. Soit $v_0$ tel que $v_0$ ait des interprétations minimales, c'est-à-dire que chaque type $u$ défini par $\delta_u$ a pour interprétation sa borne inférieure :

$$v_0(u) = \dot{\bigcup_{R \in \mathbb{A}[\delta_u]}} \widetilde{v}_0(R)$$

Ces égalités nous donnent $v_0(t)(l) \neq \Omega$.

Si $v_0(t)(l) = \tau$, avec $\tau \neq \tau_0$, alors on a directement la propriété.

Supposons que $v_0(t)(l) = \tau_0$. Montrons par récurrence sur l'ordre de définition des types que $\Gamma \vdash l$ `of` $\tau_0 \Subset t$. Il existe $R \in \mathbb{A}[\delta_u]$ tel que $\widetilde{v}_0(R)(l) = \tau_0$. Soit $R = l$ `of` $\tau_0$ et on peut conclure, soit $R = t'$, $t'$ étant un type défini avant $t$ et on peut utiliser l'hypothèse de récurrence. Donc $v_0(t)(l) = \tau_0$ implique $\Gamma \vdash l$ `of` $\tau_0 \Subset t$, ce qui est contradictoire avec les hypothèses.

Supposons donc pour finir que $v_0(t)(l) = \varnothing$. Soit $\tau_1 \in T$, avec $\tau_1 \neq \tau_0$. Soit $v_1$ la valuation :

$$v_1(u)(l) = \begin{cases} \tau_1 & \text{si } (u,l) = (t,l) \\ v_0(u)(l) & \text{sinon} \end{cases}$$

Soit $W_0$ le modèle précédant la définition de $t$. $v_1 \in W_0$ ; en effet, on n'a modifié la valuation qu'en $t$, qui n'est pas encore défini, donc on peut utiliser la propriété 1. De plus, on a :

$$v_1(t) = v_0(t) \cup \{l \mapsto \tau_1\}$$

$$v_0(t) = \dot{\bigcup_{R \in \mathbb{A}[\delta_u]}} \widetilde{v}_0(R)$$

Donc :

$$v_1(t) \supseteq \dot{\bigcup_{R \in \mathbb{A}[\delta_u]}} \widetilde{v}_1(R)$$

Donc $v_1 \in V^0_{W_0}(\delta_t)$.

Supposons qu'il existe $R \in \mathbb{C}[\Gamma(t)]$ tel que $\widetilde{v}_1(R) \cup \widetilde{v}_1(t)$ soit incohérent. Alors $\widetilde{v}_1(R)(l) \neq \varnothing$.

Supposons que $R = ?l \texttt{ of } \tau$. Quitte à choisir $\tau$ à la place de $\tau_0$ lors du choix de $\tau_0$, on peut supposer que $\tau = \tau_0$. Donc $\Gamma \vdash ?l \texttt{ of } \tau_0 \Subset t$, ce qui est contradictoire avec les hypothèses. Notons que ce changement de $\tau_0$ n'est possible que parce qu'il n'y a pas, d'après les hypothèses, deux $\tau$ différents tels que $\Gamma \vdash ?l \texttt{ of } \tau \Subset t$.

Supposons que $R = ?t'$. Donc $\widetilde{v}_1(t')(l) \neq \varnothing$. Donc il existe $\tau$ tel que $\widetilde{v}_1(t')(l) = \tau$. Par minimalité de $v_0$, on a donc $\Gamma \vdash l \texttt{ of } \tau \Subset t'$, et donc $\Gamma \vdash ?l \texttt{ of } \tau \Subset t$, ce qui nous ramène au cas précédent.

Enfin, $R \neq \neg l$ d'après les hypothèses. Donc :

$$\forall R \in \mathbb{C}[\Gamma(t)],\ \widetilde{v}_1(R) \cup \widetilde{v}_1(t) \text{ est cohérent}$$

Et donc $v_1 \in V_{W_0}(\delta_t)$. On conclut à l'aide de la propriété 2 pour montrer qu'il existe $v_2 \in W$ tel que $v_2$ soit égal à $v_1$ sur le type $t$. □

## A.6 Algorithme décidant le test de compatibilité

**Propriete 4**

*Il existe un algorithme qui, étant donnés* $\Gamma$, $\mathcal{A}_1$ *et* $\mathcal{A}_2$, *décide* $\Gamma \vdash \mathcal{A}_1 \odot \mathcal{A}_2$.

**Preuve**

Les preuves de $\Gamma \vdash \mathcal{A}_1 \odot \mathcal{A}_2$ commencent nécessairement par les règles $Disp1$ et $Disp2$, permettant de tester chaque paire d'associations. On applique une fois $Disp1$ puis un nombre fini de fois $Disp2$.

Ensuite, on peut appliquer la règle $Sym$ autant de fois qu'on veut pour chaque paire d'associations. Il est immédiat de voir que l'on peut se contenter d'utiliser au plus une fois cette règle par paire d'associations.

Ensuite on applique une règle parmis $In$, $LT1$, $LT2$, $LT3$, $LL$ ou $TT$. Quitte à faire du *backtracking*, on peut explorer chaque cas puisqu'il y a un nombre fini de ces règles.

Aucune de ces règles n'a pour prémisse une utilisation de $\odot$. On ne peut donc continuer qu'en appliquant $In1$, $In2$ ou $In3$, ou en vérifiant un axiome.

Les règles $In1$, $In2$ et $In3$ terminent ; en effet, elles ne font qu'explorer récursivement l'environnement, des types les plus récents aux plus vieux. Au pire, on termine donc en $n$ étapes, où $n$ est le nombre de définitions de types.

Enfin, les axiomes consistent soit en un test d'appartenance à l'environnement $\Gamma$, soit en une preuve d'égalité ou d'inégalité de type (ce qui dépend du système de type choisi). $\Gamma$ étant fini, le test d'appartenance est décidable. □

## A.7 Correction et complétude du test de compatibilité

**Propriete 5 (Correction et complétude)**

*A une étape* $(W, \Gamma)$ *donnée, pour toute définition de type* $\delta$, *la validité de* $\delta$ *vis à vis de* $W$ *est équivalente à la validité de* $\delta$ *vis a vis de* $\Gamma$ *et de* $\Delta$.

**Preuve**

On raisonne par récurrence sur le nombre de définitions de type.

La validité vis à vis de $\Gamma$ et de $\Delta$ :

$$\Gamma \vdash \mathbb{A}[\mathcal{T}_\Delta(\delta)] \odot (\mathbb{A}[\mathcal{T}_\Delta(\delta)] \cup !\mathbb{C}[\mathcal{T}_\Delta(\delta)])$$

est équivalente, d'après les règles $Disp1$ et $Disp2$, à la propriété suivante :

$$\forall (R, R') \in \mathbb{A}[\mathcal{T}_\Delta(\delta)] \times (\mathbb{A}[\mathcal{T}_\Delta(\delta)] \cup !\mathbb{C}[\mathcal{T}_\Delta(\delta)]),\ \Gamma \vdash R \odot R'$$

D'après la définition de $\mathcal{T}_\Delta$, cette propriété est équivalente à :

$$\begin{cases} \forall (R_1, R_2) \in \mathbb{A}[\delta]^2,\ \forall (R_1', R_2') \in \widetilde{\Delta}(R_1) \times \widetilde{\Delta}(R_2),\ \Gamma \vdash R_1' \odot R_2' \\ \forall (R_1, R_2) \in \mathbb{A}[\delta] \times \mathbb{C}[\delta],\ \forall R_1' \in \widetilde{\Delta}(R_1),\ \Gamma \vdash R_1' \odot !R_2 \end{cases}$$

Rappelons la définition de la validité vis à vis de $W$ :

$$\forall v \in W,\ \forall (R_1, R_2) \in \mathbb{A}[\delta] \times (\mathbb{A}[\delta] \cup \mathbb{C}[\delta]),\ \widetilde{v}(R_1) \cup \widetilde{v}(R_2) \text{ est cohérent}$$

Ce qui est équivalent à :

$$\begin{cases} \forall (R_1, R_2) \in \mathbb{A}[\delta]^2,\ \forall v \in W,\ \widetilde{v}(R_1) \cup \widetilde{v}(R_2) \text{ est cohérent} \\ \forall (R_1, R_2) \in \mathbb{A}[\delta] \times \mathbb{C}[\delta],\ \forall v \in W,\ \widetilde{v}(R_1) \cup \widetilde{v}(R_2) \text{ est cohérent} \end{cases}$$

Supposons que $R_1 = l_1\ \texttt{of}\ \tau_1$ et $R_2 = l_2\ \texttt{of}\ \tau_2$. $\widetilde{\Delta}(R_1) = \{R_1\}$ et $\widetilde{\Delta}(R_2) = \{R_2\}$ ; $\widetilde{v}(R_1) = \{l_1 \mapsto \tau_1\}$ et $\widetilde{v}(R_2) = \{l_2 \mapsto \tau_2\}$. Donc, par définition des règles de compatibilité :

$$\begin{array}{rcl} \Gamma \vdash R_1 \odot R_2 & \iff & l_1 \neq l_2 \text{ ou } \tau_1 = \tau_2 \\ & \iff & \forall v \in W,\ \widetilde{v}(R_1) \cup \widetilde{v}(R_2) \end{array}$$

Supposons que $R_1 = t_1$ et $R_2 = t_2$. Si $t_1$ et $t_2$ sont ouverts, alors $\widetilde{\Delta}(R_1) = \{R_1\}$ et $\widetilde{\Delta}(R_2) = \{R_2\}$. Par définition des règles de compatibilité, on a :

$$\Gamma \vdash t \odot t' \iff \Gamma \vdash ?t \Subset t' \text{ ou } \Gamma \vdash ?t' \Subset t \text{ ou } \Gamma \vdash t \Subset t' \text{ ou } \Gamma \vdash t' \Subset t$$

D'après le lemme 1, points 2 et 5, et le lemme 2, point 1, on a donc :

$$\Gamma \vdash t \odot t' \iff \widetilde{v}(t) \cup \widetilde{v}(t') \text{ est cohérent}$$

Supposons que $R_1 = t$ et $R_2 = l\ \texttt{of}\ \tau$. Si $t$ est ouvert, alors $\widetilde{\Delta}(R_1) = \{R_1\}$ et $\widetilde{\Delta}(R_2) = \{R_2\}$. Par définition des règles de compatibilité, on a :

$$\begin{array}{rl} \Gamma \vdash t \odot l\ \texttt{of}\ \tau \iff & \Gamma \vdash ?l\ \texttt{of}\ \tau \Subset t \text{ ou } \neg l \in \Gamma(t) \text{ ou } \Gamma \vdash l\ \texttt{of}\ \tau \Subset t \text{ ou} \\ & \exists \tau_1, \tau_2,\ \tau_1 \neq \tau_2,\ \Gamma \vdash ?l\ \texttt{of}\ \tau_1 \Subset t \text{ et } \Gamma \vdash ?l\ \texttt{of}\ \tau_2 \Subset t \end{array}$$

D'après le lemme 1, points 1, 3, 4 et 6, et le lemme 2, point 2, on a donc :

$$\Gamma \vdash t \odot l\ \texttt{of}\ \tau \iff \widetilde{v}(t) \cup \widetilde{v}(l\ \texttt{of}\ \tau) \text{ est cohérent}$$

Si $R_2$ est une compatibilité, on peut utiliser les mêmes preuves, puisque $!R_2$ est une association.

Pour finir, il reste à traiter le cas où $R_1$ ou $R_2$ est un nom de type fermé. Supposons que $R_1 = t$, avec $t$ fermé. $R_2$ est quelconque. Alors $\widetilde{\Delta}(R_1) = \Delta(t)$, et :

$$\widetilde{v}(R_1) = v(R_1) = \dot{\bigcup_{R \in \Delta(t)}} \widetilde{v}(R)$$

Donc, puisque $t$ est valide :

$$\begin{aligned} \widetilde{v}(t) \cup \widetilde{v}(R_2) \text{ est cohérent} &\iff \dot{\bigcup_{R \in \Delta(t)}} \widetilde{v}(R) \cup \widetilde{v}(R_2) \text{ est cohérent} \\ &\iff \forall R \in \Delta(t),\ \widetilde{v}(R) \cup \widetilde{v}(R_2) \text{ est cohérent} \end{aligned}$$

La définition de $t$ précédant la définition de $\delta$, on peut donc utiliser l'hypothèse de récurrence pour $t$ à l'étape $W_t$ :

$$\forall v \in W_t, \widetilde{v}(t) \cup \widetilde{v}(R_2) \text{ est cohérent} \iff \forall R \in \Delta(t),\ \Gamma \vdash R \odot R_2$$

Ce qui permet de conclure. □