THÈSE

présentée à l'

UNIVERSITÉ PARIS VI

pour obtenir le titre de

DOCTEUR

dans la spécialité

INFORMATIQUE

par

Christian Rinderknecht

Sujet de la thèse :

UNE FORMALISATION D'ASN.1

Application d'une méthode formelle à un langage de spécification télécom

Soutenue le 2 décembre 1998 devant le jury composé de

Professeur	**Thérèse**	**Hardin**	Président
Professeur	**Bernard**	**Lorho**	Directeurs
Docteur	**Michel**	**Mauny**	
Professeur	**Paul**	**Gloess**	Rapporteurs
Docteur	**Jacques**	**Guyard**	
Monsieur	**Olivier**	**Dubuisson**	Examinateurs
Docteur	**Hélène**	**Bachatène**	

À François Pottier.

An expedient was therefore offered, "that since words are only names for things, it would be more convenient for all men to carry about them such things as were necessary to express a particular business they are to discourse on." [...] many of the most learned and wise adhere to the new scheme of expressing themselves by things ; which has only this inconvenience attending it, that if a man's business be very great, and of various kinds, he must be obliged, in proportion, to carry a greater bundle of things upon his back, unless he can afford one or two strong servants to attend him. [...] But for short conversations, a man may carry implements in his pockets, and under his arms, enough to supply him ; and in his house, he cannot be at a loss. Therefore the room where company meet who practise this art, is full of all things, ready at hand, requisite to furnish matter for this kind of artificial converse. Another great advantage proposed by this invention was, that it would serve as a universal language, to be understood in all civilised nations, whose goods and utensils are generally of the same kind, or nearly resembling, so that their uses might easily be comprehended.

Jonathan Swift. *Gulliver's Travels*
A Voyage to Balnibarbi, chap. V.

[...] des trous, des petits trous,
encore des petits trous,
des petits trous, des petits trous,
toujours des petits trous...
Y a de quoi devenir dingue,
de quoi prendre un flingue...

Serge Gainsbourg. *Le poinçonneur des Lilas.*

Introduction

L'essor de la télématique, au carrefour de la télécommunication et de l'informatique, doit prendre en compte la diversité des architectures et des services. Les spécifications de protocoles *Open Systems Interconnection* (OSI) sont une des bases possibles pour la conception et la réalisation de systèmes informatiques hétérogènes distribués. Ces normes de communication sont publiées conjointement par les organismes internationaux ISO [1] et l'*International Telecommunication Union* (ITU), rédigés en langues véhiculaires et vernaculaires.

Le langage de spécification ASN.1 (ITU, 1994a), soit *Abstract Syntax Notation One* en anglais, est une norme conjointe de l'ISO et de l'ITU-T (secteur télécommunication de l'ITU) permettant une description des types des données susceptibles d'être échangées au cours d'une transaction entre applications. La solution apportée à l'hétérogénéité est donc symétrique : tous les participants partagent un ensemble de modules ASN.1, constituant la composante « données » du protocole, indépendamment de leur architecture locale, matérielle ou logicielle, ce qui justifie l'appellation anglaise *abstract notation*. À cet ensemble partagé par les pairs communicants est adjoint l'emploi commun d'un codage des valeurs dont les types sont définis par le protocole ASN.1, appelé *Encoding Rules* en anglais. L'ISO et l'ITU-T publient donc plusieurs codages possibles pour les valeurs de type ASN.1 : les *Basic Encoding Rules*, ou BER (ITU, 1994e), et les *Packed Encoding Rules*, ou PER (ITU, 1994f). Ces codages sont eux aussi indépendants des architectures des entités communicantes et spécifient la sérialisation de valeurs ASN.1, c'est-à-dire de la transformation de valeurs ASN.1 en suites de bits.

Chaque pair communicant soumet le protocole ASN.1 commun à un compilateur dédié à son environnement local. Les paramètres de cet environnement sont typiquement le langage servant à programmer l'application, les contraintes matérielles (comme l'architecture de la mémoire)

1. Contrairement à une idée répandue, ISO est le préfixe grec signifiant *égal* ou *même*, correct dans les trois langues officielles de l'organisme : l'anglais, le français et le russe.

et le système d'exploitation — les paramètres globaux de la communication étant le protocole ASN.1 et le codage. À partir de ces paramètres, chaque compilateur produit, d'une part, la meilleure approximation dans le langage de programmation des définitions de types ASN.1, et, d'autre part, un ensemble de couples de fonctions de codage et décodage associés à chaque type dont une valeur au moins est susceptible d'être échangée. Ces définitions vernaculaires sont finalement introduites dans la chaîne de compilation de l'application. Le fait que les définitions de type produites dans le langage de programmation de l'application soient des approximations est inévitable car les langages de programmation ne possèdent pas de langage de type aussi riche qu'ASN.1. Cela tient essentiellement au fait que les langages de programmation possèdent des fonctions, donc une bonne partie du programme est exprimée via celles-ci, et non à travers les types. Dans le cas d'ASN.1, le but est d'exprimer le plus possible sans fonctions, donc les fonctions de codage produites par le compilateur ASN.1 pour chaque type doivent effectuer des contrôles dynamiques sur les valeurs pour combler ce manque d'expressivité au niveau des types du langage de programmation.

Les codages sont spécifiés à partir de la syntaxe des valeurs ASN.1, mais en général les valeurs échangées n'existent jamais sous forme ASN.1 car elles sont dynamiquement créées par les applications, directement dans l'environnement d'exécution local, c'est-à-dire que ce sont des valeurs des types du langage de programmation et non de spécification. Ainsi, bien qu'ASN.1 permette la spécification de valeurs, celles-ci ne sont pas généralement codées et servent essentiellement au sous-typage ainsi qu'à définir des constantes du protocole, comme des valeurs par défaut.

Les compilateurs ASN.1 souffrent de non-conformité chronique par rapport à la norme X.680 (ITU, 1994a) lors de la phase d'analyse du protocole. Notre thèse est qu'une approche formelle, fondée sur des méthodes en usage en théorie de la programmation, permet de clarifier totalement X.680 et de dériver de cette recherche un analyseur de spécifications conforme à notre spécification formelle.

Les outils théoriques que nous avons employés sont ceux de la sémantique opérationnelle structurée (dite aussi sémantique naturelle), et du formalisme de preuve associé. Un des intérêts est alors que nous pouvons exprimer dans le même langage les spécifications et les algorithmes, et de fournir un cadre théorique bien connu dans lequel des preuves permettent de prouver l'équivalence de ces spécifications et de ces algorithmes (correction et complétude).

Pour réaliser notre tâche, nous avons d'abord étudié la norme ASN.1 X.680 (ITU, 1994a) avec l'aide experte de Bancroft Scott, éditeur à

l'ISO concernant ASN.1. À partir de cette compréhension empirique nous avons alors bâti une spécification en sémantique opérationnelle structurée d'ASN.1, que nous appellerons formalisation par la suite. Nous avons intégrés tous les aspects de X.680 : types, valeurs et sous-types, la syntaxe et la sémantique.

La grammaire d'ASN.1 que nous avons étudiée est celle de X.208 (ITU, 1990), car c'était la seule disponible lorsque nous avions entrepris cette tâche. La grammaire normalisée d'ASN.1 a pour but de suggérer fortement la sémantique de ses constructions, car c'est sur elle que s'appuie les explications de X.208. Malheureusement, le souci didactique rend très difficile le traitement automatique de la grammaire pour construire un analyseur syntaxique pour le langage. En effet, si l'on soumet cette grammaire au générateur d'analyseurs syntaxiques Yacc, nous obtenons des milliers de conflits décaler/réduire et réduire/réduire, ce qui implique, en faisant l'hypothèse de la correction de Yacc, que cette grammaire n'est pas LALR(1). Nous avons alors décidé de rechercher une grammaire équivalente à la première, c'est-à-dire que le langage qu'elle engendre est exactement le même, et qui soit adaptée à l'analyse syntaxique au sein même du langage OCaml (à l'aide d'un type de donnée spécial, appelé *flux*). La contrainte que nous nous sommes imposée est très forte : nous voulions une grammaire LL(1), même si OCaml permet la reconnaissance de langages engendrés par des grammaires algébriques. D'autre part, nous savions qu'il n'est pas décidable de déterminer si, étant donné deux grammaires non algébriques, les deux langages engendrés sont les mêmes ou non (A. Aho et J. Ullman, 1972). Nous savions d'autre part qu'il n'est pas décidable non plus de déterminer si, étant donné une grammaire non contextuelle, il existe une grammaire LL(1) équivalente ou non. Aussi nous avons entrepris de transformer la grammaire initiale à l'aide de pas élémentaires réversibles et bien définis, pour obtenir un grammaire dont nous prouvons qu'elle est LL(1). Chaque transformation est décrite en fonction d'opérations rationnelles (union, produit et étoile) garantissant l'invariance du langage engendré par la grammaire résultante. Les macros ASN.1 sont traitées mais pas dans toute leur généralité, étant donné que cela est irréalisable (il est indécidable de déterminer si une grammaire non algébrique est ambigüe ou non). Les restrictions apportées permettent de réaliser en OCaml un analyseur syntaxique complet, correct et mono-passe.

Notre description étant purement syntaxique, nous l'avons alors complétée par la contrepartie sémantique de X.680 qui capture l'essence des processus de codage et de décodage, sans pour autant constituer à part entière la description de la partie finale d'un compilateur ASN.1. Pour ce faire nous avons défini un codage de référence à partir des BER (ITU,

1994e), et l'ensemble des codes des valeurs ASN.1 (appelés ici *valeurs sémantiques*, par opposition aux *valeurs syntaxiques* que sont les valeurs ASN.1).

L'étape suivante à été la restructuration de la formalisation de façon à dégager un noyau d'ASN.1, c'est-à-dire un sous-ensemble de X.680, entièrement couvert par la sémantique, car la complexité de la norme ne permettait raisonnablement pas de d'étendre la sémantique à toutes les spécifications possibles. Le noyau d'ASN.1 est ainsi défini de façon opérationnelle au moyen d'une suite de réécritures à partir de X.680. Le but fondamental de ces réécritures est de réduire les très nombreuses constructions de X.680 et d'obtenir des spécifications plus simples.

Nous avons ensuite formalisé le contrôle syntaxique des types du noyau d'ASN.1, c'est-à-dire, étant donné une valeur et un type, déterminer s'ils sont compatibles (si la valeur est bien typée). Nous énonçons alors et prouvons le théorème suivant : si l'on impose aux types une contrainte supplémentaire et normalisée dans X.680, à savoir que les noms des champs des types structurés doivent être tous distincts entre eux, alors notre spécification du contrôle des types est non-ambigüe, ou, en termes de l'algorithme associé : ce dernier est déterministe. Cela signifie que le contrôle des types, s'il réussi, ne peut le faire que d'une seule façon. Cela implique aussi que l'analyse par cas de notre spécification peut être reprise telle quelle pour écrire l'algorithme de contrôle syntaxique des types.

Toujours en restant au niveau syntaxique, nous avons complété le contrôle des types par le contrôle des sous-types, c'est-à-dire, étant donné un type T et une contrainte de sous-typage σ, déterminer s'ils sont compatibles, c'est-à-dire si le sous-type résultant possède au moins une valeur. Supposons la donnée d'une déclaration de valeur d'un sous-type. Nous commençons par réduire la spécification pour l'amener dans le noyau d'ASN.1, en effectuant un certain nombre de contrôles. Ensuite, nous ignorons les contraintes de sous-typage et nous appliquons le contrôle des types que nous venons juste d'esquisser. La dernière étape est de s'assurer, sachant que la valeur v est de type T, si le type T contraint par σ et v, est un sous-type non vide. La raison de ce procédé en deux temps (type, puis sous-type plus valeur) est la grande dépendance entre types, valeurs et contraintes de sous-typage, qui rend improbable l'existence d'un traitement global compréhensible et correct si le contrôle des sous-types a une contre-partie sémantique. Le principe du contrôle des sous-types est fondé sur une série de réécritures qui sont censées laisser invariante la sémantique. Les contraintes complexes sont réduites en des contraintes élémentaires, comme X.680 est réduit vers le noyau d'ASN.1.

Nous définissons ensuite la sémantique du noyau. Cette partie est

constituée d'abord de la définition des codes et d'un codage de référence pour les valeurs ASN.1. Puis nous définissons un contrôle sémantique des types, c'est-à-dire, étant donné un code et un type du noyau, déterminer s'ils sont compatibles (si le code est bien typé). À partir de ces relations, nous énonçons et prouvons un théorème de correction sémantique du codage de référence dans le noyau : soient une valeur v de type T (contrôle syntaxique des types) et c le code de v en supposant que v est de type T (codage de référence), alors le code c est de type T (contrôle sémantique des types). L'interprétation informelle de ce résultat de correction de la transmission est d'établir formellement que le type des valeurs ASN.1 dans le noyau est conservé par le codage de référence, ce qui constitue la quintessence des BER. Le chapitre consacré à la sémantique du noyau est complété par un théorème supplémentaire ayant trait à la non-ambigüité de la transmission. Nous énonçons et démontrons qu'en imposant aux types une contrainte supplémentaire et normalisée par X.680, portant sur les étiquettes des champs des types structurés, nous pouvons prouver que notre spécification du contrôle sémantique des types est non-ambigüe, ou, en termes de l'algorithme associé, que ce dernier est déterministe. Cela signifie que le contrôle sémantique des types, s'il réussi, ne peut le faire que d'une seule façon. En d'autres termes encore, cela implique qu'un éventuel décodage fondé sur ce contrôle s'exécutera sans rebroussements, donc efficacement. Cela implique aussi que l'analyse par cas de notre spécification formelle peut être reprise telle quelle pour écrire l'algorithme de contrôle sémantique des types.

En regroupant tous nos théorèmes, nous pouvant dire que nous avons établi, pour le noyau d'ASN.1 dont les types structurés sont contraints sur les noms et les étiquettes de leurs champs, le résultat global suivant. Soit c le code d'une valeur v de type T. Alors un décodage de c, fondé sur notre contrôle sémantique des types, produirait une valeur ASN.1 identique (soit syntaxiquement égale) à v. En d'autres termes, la composition du codage et du décodage est l'identité. Dans la pratique toute fois, le décodage ne produit jamais de valeurs ASN.1, car celles-ci n'existent que statiquement, mais exprimer ainsi notre résultat en montre mieux la portée.

Rappelons que la réduction de X.680 vers le noyau et le contrôle des sous-types sont des étapes purement syntaxiques et non couvertes par la sémantique, à cause de la très grande complexité de X.680 qui aurait rendu probablement impossible l'établissement des preuves par un humain (combinatoire astronomique des cas).

Le plan de ce document est le suivant. En annexe se trouvent la formalisation complète, les preuves de nos théorèmes ainsi que l'étude de la syntaxe d'ASN.1. Cette annexe intéressera les théoriciens des langages

qui y trouveront des définitions rigoureuses et précises des concepts, formalismes, spécifications et algorithmes en jeu. Les chapitres qui suivent présentent de façon informelle notre thèse à l'aide de nombreux exemples en syntaxe ASN.1, et sont destinés aussi bien aux télécommunicants qu'aux amateurs de logique formelle. Tout d'abord le chapitre 1 page ci-contre est consacré à l'obtention d'un noyau de X.680. Le chapitre 2 page 37 présente le contrôle syntaxique des types. Le chapitre 3 page 45 traite de la sémantique. Pour finir, le chapitre 4 page 63 présente le contrôle des sous-types.

Chapitre 1

Noyau

Ce chapitre a pour objectif de présenter la première phase nécessaire à l'analyse d'un module ASN.1. Nous allons nous livrer à une transformation du module de façon à nous assurer d'un certain nombre de propriétés prévues par la norme et à faciliter ainsi grandement les traitements ultérieurs présentés dans les chapitres suivants. Ce chapitre possède une contrepartie formelle en annexe pour les lecteurs ayant des connaissances en théorie de la programmation (section 6 page 119). À chaque étape de ce chapitre nous renverrons ces lecteurs à l'étape formelle associée.

Dans une optique de compilation, la présente transformation peut être assimilée à une précompilation des sources ASN.1, qui détecte un certain nombre d'écarts à la norme et simplifie les phases suivantes, comme le contrôle des types, le codage, et même le décodage. Nous appellerons *noyau ASN.1* l'ASN.1 résultant de ce processus de précompilation.

Mais pourquoi une telle étape de mise en forme est-elle nécessaire ? Fondamentalement, c'est la très grande dépendance mutuelle entre les différents concepts du langage qui l'impose. Nous pouvons, par une première analyse, décomposer ASN.1 en trois concepts : les types, les sous-types et les valeurs. La syntaxe nous apprend que les types dépendent des valeurs (via les valeurs par défaut pour les champs de structures, `DEFAULT`, ou les constantes distinguées du type `INTEGER` par exemple) et dépendent aussi des sous-types (les types structurés possèdent des champs qui peuvent contenir des sous-types). Similairement, les sous-types dépendent des types via les inclusions de types dans les contraintes `INCLUDES` et des valeurs (le sous-typage d'ASN.1 étant ensembliste, une contrainte spécifie ultimement des valeurs que doit *contenir* un type). Les valeurs quant à elles dépendent indirectement des types en ce sens qu'elles sont toutes spécifiées conjointement avec leur type, et que leur

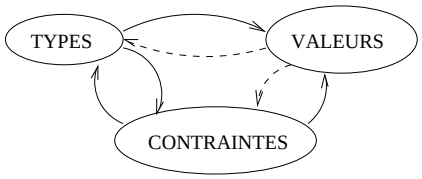

FIGURE 1.1 – Graphe de dépendance entre concepts ASN.1

interprétation en dépend fortement, par exemple via l'emploi des constantes distinguées du type `INTEGER` ou des constantes énumérées du type `ENUMERATED`. Pour résumer la situation, nous donnons le graphe de dépendance entre types, valeurs et contraintes à la figure 1.1. Les arcs pleins sont les dépendance entre termes, c'est-à-dire entre les extraits de syntaxe produits par les règles de grammaire `Type`, `Value` et `Constraint`, et les arcs en pointillés sont les dépendances supplémentaires en tenant compte des déclarations : la déclaration d'une valeur implique syntaxiquement la présence d'un type et éventuellement d'une contrainte. Il est évident que ce graphe est complet. Le lecteur ayant une connaissance du langage OCaml et de notions de compilation est invité à se référer au chapitre 5 page 97 pour les détails théoriques d'un telle présentation (syntaxe abstraite et environnements).

Pour décrire la transformation des spécifications ASN.1, nous étendrons un peu la syntaxe du langage pour être à même de définir une réécriture, c'est-à-dire la réduction d'un langage vers un sous-langage strict. Ainsi nous n'aurons pas besoin de sortir de ce cadre purement syntaxique pour en parler (pour les lecteurs familiers avec l'informatique fondamentale, cela veut dire que cette étape ne possède pas de contrepartie sémantique). Cette réécriture se fera en étapes successives, dont l'ordre doit être respecté car ces étapes ne commutent pas entre elles.

Pour commencer, il convient de dire quelques mots concernant le type `REAL` : ce sera la **première étape**. Ce type est défini dans une norme (ITU, 1994a, § 18), mais sa définition n'en est pas vraiment une. En effet, il est dit (ITU, 1994a, § 18.5) qu'on lui *associe* la définition

```
SEQUENCE {
  mantissa INTEGER,
  base     INTEGER (2|10),
  exponent INTEGER
}
```

et qu'il possède en plus la particularité de contenir trois valeurs spécialement distinguées (ITU, 1994a, § 18.6) : `0`, `PLUS-INFINITY` et `MINUS-INFINITY`, *toutes trois ne pouvant être définies en termes du type* associé *ci-dessus*. Pour clarifier cette situation, nous avons pris la décision dans cette thèse de définir implicitement

```
REAL ::= SEQUENCE {
           mantissa INTEGER,
           base     INTEGER (2|10),
           exponent INTEGER
         }
```

et de toujours traiter spécialement les cas des valeurs spéciales ci-dessus, lorsqu'elles se présentent, par exemple lors du contrôle des types. De cette façon, nous éliminons le concept indéfini d'association et nous établissons clairement quels sont les cas exceptionnels. Le fait que la définition que nous adoptons soit dite implicite signifie qu'elle préexiste au module que nous traitons et en fait toujours partie automatiquement. `REAL` reste un mot-clé, donc notre définition n'entrera jamais en conflit lexical avec les définitions du spécifieur de protocoles. L'ajout de notre définition implicite au module courant est formalisé à la section 6.1 page 119.

Avant d'aborder la seconde étape, indiquons le sens que nous entendons donner à certains termes. Tout d'abord une *étiquette* est une annotation de type définie en (ITU, 1994a, § 28.1) par la règle de grammaire `Tag`. Nous nommerons *étiquetage* une succession d'étiquettes, éventuellement séparées par les *attributs d'étiquettes* `IMPLICIT` ou `EXPLICIT`. Par exemple, dans

```
T ::= [PRIVATE 4] IMPLICIT BOOLEAN
```

nous avons une étiquette `[PRIVATE 4]` dont l'attribut est `IMPLICIT`. Le concept d'étiquetage, quant à lui, n'apparaît pas explicitement dans la pratique, mais il est utile pour notre présentation. Soit

```
T ::= [PRIVATE 4] IMPLICIT BOOLEAN
U ::= [5] EXPLICIT T
```

Le type `U` est une *abréviation* du type `T`, c'est-à-dire que sa définition ci-dessus est équivalente à

```
U ::= [5] EXPLICIT [PRIVATE 4] IMPLICIT BOOLEAN
```

Cette autre définition de `U` est valide, mais il est très peu probable qu'elle apparaisse dans la pratique. Dans ce cas, nous dirons que le type `BOOLEAN` (et non le type `U`) possède un étiquetage qui vaut

```
[5] EXPLICIT [PRIVATE 4] IMPLICIT
```

et qui est constitué de deux étiquettes, dans l'ordre attribuées `IMPLICIT` et `EXPLICIT` (ordre inverse d'écriture).

La **deuxième étape** (section 6.2 page 119) de la mise en forme des modules ASN.1 consiste à définir un concept de *type bien fondé* et à s'assurer que tous les types de la spécification sont bien fondés. Ce

concept a principalement pour racine la difficulté que présente la définition des types récursifs. La seule indication au sujet des définitions récursives (ITU, 1994a, § 3.8.42) dit que la validité de telles constructions est laissée au jugement du spécifieur de protocoles, la seule contrainte étant que les valeurs de ce type aient un codage fini. Dans cette thèse nous définissons formellement un codage (section 8.1 page 159) qui est inspiré des *Canonical Encoding Rules*, ou CER (ITU, 1994e). Pour guider l'intuition, dans ce chapitre le lecteur pourra néanmoins entendre *CER* lorsqu'il lira *codage*. Ajoutons que les compilateurs ASN.1 présentent de nombreuses déficiences liées aux traitements des types et des valeurs récursives. Il s'agit d'empêcher par exemple

— les définitions de types dont toutes les valeurs ont un codage infini. Par exemple :
```
T ::= SET {a [0] T}
```
En effet, `T` ne permet que des définitions de valeurs de la forme :
```
x T ::= {a x}
y T ::= {a {a y}}
z T ::= {a {a {a z}}}
```
Toutes les valeurs de `T` ont un codage infini, ce qui signifie en pratique que le processus de codage ne termine pas en présence de telles valeurs.

— les déclarations qui ne définissent pas un type unique. Par exemple :
```
T ::= T
```
Ici nous avons affaire à une déclaration qui n'est pas une définition. Nous nommons *déclaration* un texte qui peut être produit par la règle de grammaire `TypeAssignement` (ITU, 1994a, § 13.1). Une *définition*, par contre, est une déclaration qui vérifie un certain nombre de critères de validité. L'analogie mathématique est qu'une déclaration est la donnée d'une équation dont la syntaxe est correcte, et la définition est l'existence d'une solution de cette équation sous certaines conditions, par exemple, l'unicité. Ainsi, dans l'exemple ci-dessus, la déclaration `T ::= T` n'est pas suffisamment contrainte pour assurer l'unicité d'une solution `T` : tout type peut être défini de la sorte. Nous devons donc rejeter cette déclaration.

— les définitions de types dont certaines valeurs n'ont pas d'étiquetage (rappelons que les étiquetages participent au codage des valeurs). Soit par exemple :
```
T ::= CHOICE {
        a T,
        b [0] INTEGER
      }

t T ::= a : t
```

Le codage de la valeur `t` échouera car on ne peut lui associer un étiquetage (nous laissons pour l'instant de côté le fait que la valeur est récursive elle aussi).

Remarquons que les champs des types structurés pouvaient ne pas avoir de label (ITU, 1990), et que cette dissociation entre label et étiquetage pouvait mener à la déclaration de types dont certaines valeurs avaient une infinité d'étiquetages. Par exemple :

```
T ::= CHOICE {
          [0] T,
        b [1] INTEGER
      }
t T ::= b : 7
```

Le codage de la valeur `t` peut être associé à une infinité d'étiquetages : `[1]`, `[0][1]`, `[0][0][1]`, etc.

Le concept de type bien fondé permet d'écarter toutes les déclarations que nous avons énumérées. Mais que faut-il alors penser des déclarations suivantes :

- `T ::= SET OF T`
- `T ::= SET {a T OPTIONAL}`
- `T ::= SET {a T DEFAULT t}`
- `T ::= SET {COMPONENTS OF T}`
- `T ::= CHOICE {a [0] T}`
- `T ::= CHOICE {a [0] a < T}`
- `T ::= CHOICE {a [0] T, b NULL} (WITH COMPONENTS {b ABSENT})`
- `T ::= INTEGER (INCLUDES T)`

Envisageons-les dans l'ordre :

```
T ::= SET OF T
```

Il existe une infinité de valeurs de ce type dont le codage est infini :

```
x T ::= {x}
y T ::= {{y}}
z T ::= {{{z}}}
```

Mais il possède une unique valeur dont le codage est fini :

```
t T ::= {}
```

Nous dirons que les types `SET OF` et `SEQUENCE OF` sont toujours bien fondés, car la valeur `{}` est toujours possible. Notons qu'il n'est pas nécessaire d'imposer que le type des éléments soit bien fondé, par exemple si `A` n'est pas bien fondé, alors `SET OF A` est bien fondé.

Que penser alors de

```
T ::= SET {a T OPTIONAL}
```

Les clauses (ITU, 1994a, § 22.8) et (ITU, 1994a, § 24.8) nous permettent d'écrire :

```
t T ::= {}
```

Là encore, cette valeur valide du type `T` est unique et il existe une infinité de valeurs dont le codage est infini :

```
x T ::= {x}
y T ::= {{y}}
z T ::= {{{z}}}
```

Que dire si le champ `a` dans l'exemple précédent avait été annoté `DEFAULT` ?

```
T ::= SET {a T DEFAULT t}
```

Ici encore, il existe une infinité de valeurs dont le codage est infini et nous pouvons écrire *a priori* :

```
t T ::= {}
```

Mais la clause (ITU, 1994a, § 22.9) implique que si la réécriture

```
t T ::= {t}
```

est invalide, alors la première aussi. Or cette dernière doit être rejetée, comme nous l'avons dit précédemment, donc il faut rejeter la première.

Considérons maintenant

```
T ::= SET {COMPONENTS OF T}
```

Ici, nous avons une déclaration récursive à travers la construction `COMPONENTS OF`. Nous n'avons à aucun endroit la donnée des champs à inclure, donc nous déciderons que, si un type est bien fondé, alors il ne présente pas une telle récursivité. Considérons ensuite

```
T ::= CHOICE {a [0] T}
```

Il existe une infinité de valeurs pour ce type et elles sont de la forme suivante :

```
x T ::= a : x
y T ::= a : a : y
z T ::= a : a : a : z
```

Toutes ces valeurs auraient un étiquetage différent associé lors du codage, mais le fait qu'elles soient toutes récursives implique que le processus de codage ne termine pas. Nous aborderons la récursivité des valeurs plus loin. Nous décidons en tout cas ici que le type `T` ci-dessus n'est pas bien fondé.

Faisons intervenir maintenant les types sélectionnés (nous parlerons de *sélection*) :

```
T ::= CHOICE {a [0] a < T}
```

Les seules valeurs possibles sont de la forme suivante :

```
x T ::= a : x
y T ::= a : a : y
z T ::= a : a : a : z
```

Par rapport au cas précédent, ces valeurs n'ont pas d'étiquetage et, de toute façon, leur récursivité suffit à les disqualifier, ainsi que le type qui les engendre, car ce sont ses seules valeurs possibles. Pour comprendre ce point, ouvrons ici une parenthèse au sujet des sélections, définies en (ITU, 1994a, § 27). Il n'est en effet pas clair *a priori* si la spécification

```
A ::= [0] i < [1] B
B ::= [2] CHOICE {i [3] REAL}
```

équivaut à

```
A ::= [0] [3] REAL
B ::= [2] CHOICE {i [3] REAL}
```

En fait c'est bien ainsi qu'il faut interpréter la norme. Donc la particularité de la sélection est qu'elle ignore toutes les étiquettes sauf celle du champ sélectionné. C'est pourquoi dans notre exemple `T ::= CHOICE {a [0] a < T}`, l'étiquette ne sert pas au codage des valeurs de `T`. Examinons ensuite

```
T ::= CHOICE {a [0] T, b NULL} (WITH COMPONENTS {b ABSENT})
```

Ici intervient une contrainte de sous-typage, définie en (ITU, 1994a, § 45.8). Cette déclaration est équivalente à

```
T ::= CHOICE {a [0] T}
```

et nous avons déjà conclu que ce type n'était pas bien fondé. Mais cette réécriture des contraintes de sous-typage fait l'objet d'une phase à part entière (chapitre 9 page 217), qui intervient après la mise en forme liminaire que nous sommes en train de décrire ici. Pour le moment, nous ignorerons donc la contrainte de sous-typage et conclurons que le type `CHOICE {a [0] T, b NULL}` est bien fondé (ce qui est absolument vrai) et, après la réécriture des contraintes de sous-typage, nous devrons à nouveau vérifier que les types réécrits sont bien fondés. Cette vérification en deux temps semble bien compliquée, mais la réécriture des contraintes de sous-typage est extrêmenent complexe et il est de loin préférable de procéder ainsi, plutôt que de chercher à définir une seule phase englobant tout. Pour résumer : nous acceptons temporairement cette déclaration, en attendant la phase de réécriture des contraintes de sous-typage après laquelle nous réappliquons notre critère de bonne fondation.

Il ne nous reste qu'à envisager notre dernier exemple, lui aussi impliquant une contrainte de sous-typage :

```
T ::= INTEGER (INCLUDES T)
```

La contrainte `INCLUDES` est décrite en (ITU, 1994a, § 45.3). Ici, la récursivité s'effectue à travers cette contrainte, qui est la seule qui le permette. Rappelons, à travers un exemple simple le sens d'une telle contrainte :

```
Day ::= ENUMERATED {monday (0), tuesday (1), wednesday (2),
             thursday (3), friday (4), saturday (5), sunday (6)}

Week-End ::= Day (saturday | sunday)

Long-Week-End ::= Day (INCLUDES Week-End | monday)
```

Bien que (ITU, 1994a, § 45.3.2) ne dise pas les choses de cette manière, nous dirons dans cette thèse que la sémantique de `INCLUDES` implique que `Long-Week-End` doit être réécrit en

```
Long-Week-End ::= Day (saturday | sunday | monday)
```

Il faut aussi, comme le dit cette fois explicitement (ITU, 1994a, § 45.3.2), que `Week-End` et `Day` aient même sur-type, soit le même type à partir duquel ils sont produits par sous-typages successifs. Quel est alors le problème avec notre déclaration ci-dessous ?

```
T ::= INTEGER (INCLUDES T)
```

D'après (ITU, 1994a, § 45.3.2), il n'y a aucun problème, car `INTEGER` et `T` sont dérivés du même type prédéfini `INTEGER`. Cependant, cette clause n'est pas assez restrictive. En effet, de façon ensembliste (et non opérationnelle, comme nous l'avons fait précédemment) une telle déclaration signifie que nous avons l'équation :

$$\mathcal{S}(\texttt{T}) = \mathcal{S}(\texttt{INTEGER}) \cap \mathcal{S}(\texttt{T}),$$

où $\mathcal{S}(x)$ désigne l'ensemble (au sens mathématique) des valeurs du type x. Il est évident que cette équation est satisfaite pour tout sous-type, c'est-à-dire sous-ensemble, de `INTEGER`, donc ne possède pas de solution unique. Pour cette raison, nous devons rejeter cette déclaration, mais nous ne le ferons pas ici, car notre relation de type bien fondé n'examine pas les contraintes de sous-typage. Nous accepterons donc `T` comme bien fondé, jusqu'à ce que la phase de contrôle des sous-types le rejette. Le lecteur attentif remarquera qu'il y a unicité si, au lieu de `INTEGER`, nous avions eu un type qui ne contienne qu'une valeur. Par exemple :

```
T ::= NULL (INCLUDES T)
```

Il est exact dans ce cas que cette déclaration a un sens : elle est équivalente à

```
T ::= NULL
```

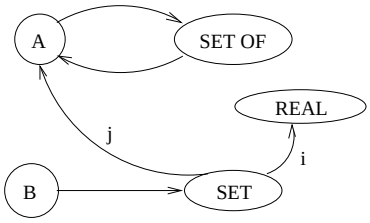

FIGURE 1.2 – Graphe modélisant des déclarations de type

car `NULL` est la seule solution non vide à l'équation

$$\mathcal{S}(\texttt{T}) = \mathcal{S}(\texttt{NULL}) \cap \mathcal{S}(\texttt{T}).$$

Il serait néanmoins très compliqué de détecter ces cas, et nous nous tiendrons donc à notre résolution de les rejeter en bloc lors de la phase de contrôle des sous-types.

Après avoir étudié divers cas extrêmes, tentons maintenant de dégager une définition informelle de notre critère de bonne fondation. Pour cela, nous proposons de concevoir les déclarations de types comme la donnée d'un graphe. Chaque nom de type apparaissant à gauche du symbole `::=` est un sommet et chaque type à droite de ce même symbole aussi. De même, chaque type apparaissant dans d'autres types est un sommet. Par exemple :

```
A ::= SET OF A
B ::= SET {
        i REAL,
        j A
      }
```

définit le graphe à la figure 1.2

Avec ce modèle intuitif en tête, nous définissons alors le prédicat « *est bien fondé* » par cas comme suit :

- Un type non récursif est bien fondé. (Un type non récursif correspond à un graphe sans circuit.)
- Un type récursif est bien fondé si et seulement si il existe le long de chaque circuit du graphe associé
  - soit un sommet `SET OF` ou `SEQUENCE OF` ;
  - soit un sommet `SET` ou `SEQUENCE` dont l'arc sortant sur le circuit est annoté `OPTIONAL` ;
  - soit un sommet `CHOICE` dont l'arc entrant ne provient pas d'un sommet `<`, même à travers des arcs d'abréviation, *et* dont au moins un des arcs sortants n'appartient pas à un circuit.

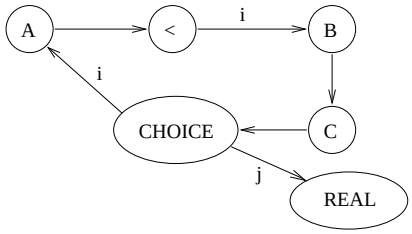

FIGURE 1.3 – Graphe mal fondé

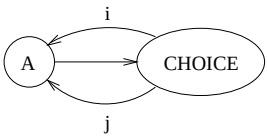

FIGURE 1.4 – Un autre graphe mal fondé

Nous pensons qu'il est utile de détailler le troisième point du second cas, soit le sommet `CHOICE`. La condition est une conjonction. Tout d'abord, qu'est-ce qu'un arc entrant provenant d'un sommet `<` même à travers des arcs d'abréviation ? Prenons un exemple :

```
A ::= i < B
B ::= C
C ::= CHOICE {
        i A,
        j REAL
      }
```

Cette spécification est mal fondée car elle s'interprète en le graphe à la figure 1.3. Un arc d'abréviation est un arc correspondant à une abréviation de type, ici : `B ::= C`. Le sommet `CHOICE` de notre exemple possède donc un arc entrant qui correspond à une abréviation et l'arc précédant sur le circuit provient d'un sommet `<`. C'est ce que nous interdisons dans la première partie de la conjonction du troisième cas des types récursifs.

Que signifie maintenant le second terme de la conjonction : un sommet `CHOICE` dont au moins un des arcs n'appartient pas à un circuit ? Prenons un exemple :

```
A ::= CHOICE {
        i [0] A,
        j [1] A
      }
```

Le type `A` n'est pas bien fondé car il correspond au graphe à la figure 1.4. Tous les arcs sortants du sommet `CHOICE` sont sur un circuit. Donc toutes

les valeurs de ce type auront un codage infini (condition suffisante), ce qu'il faut interdire. Voilà pour le cas du sommet `CHOICE` et de notre définition de la bonne fondation des types.

Terminons en soulignant l'importance de la bonne fondation des types. En effet, même si elle n'empêche pas la définition de valeurs récursives, *elle permet d'écarter les types dont toutes les valeurs ont un codage infini*, donc qui ne termine pas. Ce critère est donc intimement en relation avec le codage et ses propriétés. Rappelons aussi que les contraintes de sous-typage ne sont pas prises en compte ici, en attendant la fin de la phase qui les traite (chapitre 9 page 217), pour réappliquer ensuite notre critère de bonne fondation — ceci dans le but de découpler au maximum les différentes phases d'analyse.

La **troisième étape** (section 6.3 page 122) s'inscrit dans la lutte contre la dépendance mutuelle entre types, valeurs et contraintes de sous-typage, que nous avons déjà évoquée. Nous commençons par découpler les valeurs globales, c'est-à-dire qui ont un nom, introduit à gauche d'un symbole `::=` de leur type, dans le but de parvenir à une spécification sur laquelle nous pourrons traiter séparément les types, les valeurs et les contraintes de sous-typage. Par exemple

```
x [0] SET {a INTEGER} (WITH COMPONENTS {a (0|1)}) ::= {a 0}
```

est réécrite en

```
x T ::= {a 0}
T ::= [0] SET {a INTEGER} (WITH COMPONENTS {a (0|1)})
```

Ainsi les déclarations de valeurs ne dépendent de leur type, éventuellement étiqueté et sous-typé, qu'à travers une abréviation, ici `T`. Nous devons donc produire un nom de type qui n'entre pas en collision avec les noms des types présents initialement et aussi qui n'entrera jamais en conflit avec les futurs noms engendrés. Pour illustrer ce fait, nous étendons le lexique d'ASN.1 en admettant qu'un nom de type puisse débuter par le caractère `#`, tout en interdisant au spécifieur cette possibilité (qui n'est pas permise par la norme). Ainsi, nous réécrirons à l'avenir :

```
x #1 ::= {a 0}
#1 ::= [0] SET {a INTEGER} (WITH COMPONENTS {a (0|1)})
```

Ce choix de notation est proche de ce que ferait un compilateur en interne à l'aide d'un générateur de symboles globaux et uniques. Nous ferons suivre le symbole `#` d'un nombre entier unique le temps de l'analyse.

La **quatrième étape** consiste à découpler les contraintes de sous-typage et les types (nous venons de découpler les valeurs globales et leur type). La syntaxe nous dit que la seule dépendance possible est à travers l'emploi de la contrainte `INCLUDES`. Par exemple :

```
T ::= SET (INCLUDES [0] SET OF INTEGER (0|1)) OF INTEGER
```

Cet usage de `INCLUDES` est exotique et hautement improbable en pratique, mais nous devons néanmoins l'envisager si nous voulons un modèle complet d'ASN.1, car un modèle s'intéresse au possible et non au probable. Ainsi nous réécrirons (section 6.4.2) :

```
T  ::= SET (INCLUDES #1) OF INTEGER
#1 ::= [0] SET OF INTEGER (0|1)
```

On notera que, contrairement à la règle générale, les éventuelles étiquettes du types inclus, soit celui introduit par `INCLUDES`, seront déplacées avec la nouvelle abréviation. Pour les lecteurs de la contrepartie formelle, cette étape apparaît juste après la cinquième étape qui suit, ceci pour des raisons très techniques. Néanmoins, la présentation de ce chapitre est la plus intuitive et elle doit guider la compréhension.

La **cinquième étape** consiste à réécrire les types structurés, c'est-à-dire construits à partir d'autres types, de telle sorte que les types résultants ne contiennent, au sens syntaxique, que des abréviations de type non contraintes (section 6.4.1). En d'autres termes, il s'agit de donner un nom à tous les types qui apparaissent dans la définition d'autres types (on dira aussi *rendre globaux les types internes*), et de rendre globales les contraintes de ces types, c'est-à-dire que toutes les contraintes apparaissent au niveau le plus haut dans leur déclaration. Commençons par :

```
A ::= SET OF SET OF SET {a REAL}
```

qui est réécrite en

```
A  ::= SET OF #1
#1 ::= SET OF #2
#2 ::= SET {a #3}
#3 ::= REAL
```

Rappelons que dans cette thèse le type `REAL` est une abréviation, mais elle est néanmoins extraite du type `#2`. Un autre exemple est

```
A ::= SET {COMPONENTS OF [1] SET {i [0] REAL}}
```

qui devient

```
A  ::= SET {COMPONENTS OF [1] #1}
#1 ::= SET {i [0] #2}
#2 ::= REAL
```

Cet usage de `COMPONENTS OF` est plutôt exotique et fort improbable en pratique, mais nous devons le prendre en compte pour faire ressortir la règle sous-jacente qui nous guide. Et si nous avions eu

```
A ::= SET OF SET OF SET {a REAL (0 | PLUS-INFINITY)}
```

alors nous aurions réécrit en

```
A  ::= SET OF #1
#1 ::= SET OF #2
#2 ::= SET {a #3}
#3 ::= REAL (0 | PLUS-INFINITY)
```

L'idée est de « remonter » les contraintes éventuelles des types internes, soit celles servant à bâtir d'autres types, au niveau des déclarations. Mais si les types internes sont étiquetés, alors ces étiquettes restent en place :

```
A ::= SET {
        a [0] SET OF [2] REAL,
        b [1] INTEGER (0|1)
      }
```

est réécrite en

```
A  ::= SET {
         a [0] #1,
         b [1] #2
       }
#1 ::= SET OF [2] #3
#2 ::= INTEGER (0|1)
#3 ::= REAL
```

Rappelons que tous les types internes sont déplacés et remplacés par une abréviation, même si ce sont déjà des abréviations. Ce n'est pas une optimisation que de ne pas les déplacer : nous avons besoin de la propriété que les abréviations internes sont uniques — voir quatorzième étape. Ce choix est lié, comme nous le verrons plus loin, à la raison qui nous oblige à laisser en place les étiquettes, qui est liée aux transformations qui leur sont spécifiques, et qui auront lieu plus tard. Remarquons de plus que l'étape que nous venons de décrire doit avoir lieu après celle consistant à découpler les valeurs globales de leur type, et celle qui découple les contraintes des types, via `INCLUDES` : après ces dernières toutes les réécritures sur les types portent bien sur tous les types du module ASN.1.

La **sixième étape** (section 6.4.3 page 125) consiste à déplier les abréviations et sélections globales. Le dépliage est l'opération qui consiste à remplacer une abréviation ou une sélection par sa définition — on parle de dépliage *partiel* si la définition est elle-même une abréviation ou une sélection, sinon on le dit *complet*. Par exemple, négligeons l'étape précédente (la cinquième), et supposons que nous ayons :

```
A ::= i < B
B ::= C
C ::= CHOICE {i D}
D ::= INTEGER
```

Alors cette spécification est réécrite en

```
A ::= INTEGER
B ::= CHOICE {i D}
C ::= CHOICE {i D}
D ::= INTEGER
```

Cette transformation a donc pour effet d'éliminer les déclarations d'abréviations et de sélections. Qu'est-ce qui nous motive ? Tout d'abord, les sélections sont des types très difficiles à parcourir car ils nécessitent de conserver la trace des noms des champs sélectionnés (ici `i`), jusqu'à trouver un `CHOICE` qui nous permet d'en éliminer un, et ainsi de suite, jusqu'à trouver un type qui ne soit ni une sélection ni une abréviation. Ce procédé est fastidieux et c'est pourquoi nous préférons nous débarrasser une fois pour toute de toutes les sélections de notre spécification (elles sont en effet toutes au niveau des déclarations, grâce à l'étape précédente). Il est aussi commode de ne pas avoir de déclarations d'abréviations, et donc de seulement les conserver comme types internes, soit ceux servant à construire d'autres types — puisque l'étape précédente a posé des abréviations partout à la place des types internes. Remarquons que notre démarche bien ordonnée nous permet d'éviter l'écueil qui toujours menace dans l'analyse d'un module ASN.1 : la non-terminaison due aux dépendances mutuelles. Par exemple, le fait que tous nos types soient bien fondés nous permet de déplier nos abréviations sans crainte de boucles. Idem pour les sélections. Voyons ce qu'il advient alors d'un type récursif bien fondé :

```
A ::= CHOICE {
        i [0] A (WITH COMPONENTS {..., j (0)}),
        j [1] INTEGER
      } (WITH COMPONENTS {..., j (0|1)})
```

Cette déclaration est réécrite à la cinquième et précédente étape en :

```
A  ::= CHOICE {
         i [0] #1,
         j [1] #2
       } (WITH COMPONENTS {..., j (0|1)})

#1 ::= A (WITH COMPONENTS {..., j (0)})

#2 ::= INTEGER
```

Maintenant nous produisons

```
A  ::= CHOICE {
         i [0] #1,
         j [1] #2
       } (WITH COMPONENTS {..., j (0|1)})
```

```
#1 ::= CHOICE {
         i [0] #1,
         j [1] #2
       } (WITH COMPONENTS {..., j (0|1)})
         (WITH COMPONENTS {..., j (0)})

#2 ::= INTEGER
```

Pour conclure cette sixième étape, il est très important de noter qu'ASN.1 viole la propriété de *substitutivité*, c'est-à-dire que si l'on remplace directement une abréviation par sa définition nous pouvons modifier la sémantique de la spécification — ce qui est inadmissible. Est-ce à dire que ce qui précède est invalidé ? Non. Le cas pathologique qui doit être traité spécialement est celui des types `SET OF` et `SEQUENCE OF` dans le cas où une contrainte peut leur être appliquée en même temps qu'elle peut l'être au type des éléments qu'ils spécifient. Reprenons l'exemple donné par Steedman (1990) :

```
A ::= SET OF PrintableString
B ::= A (SIZE (7))
```

Nous ne pouvons pas le réécrire en

```
A ::= SET OF PrintableString
B ::= SET OF PrintableString (SIZE (7))
```

En effet, dans le premier cas la contrainte `SIZE` s'applique au type `SET OF`, alors que, dans le deuxième, elle s'applique à `PrintableString`. (Les théoriciens des langages parlent alors de *capture*.) Nous traiterons donc spécialement ce cas en réécrivant ainsi :

```
A ::= SET OF PrintableString
B ::= SET (SIZE (7)) OF PrintableString
```

La **septième étape** (section 6.4.4 page 126) traite des insertions de champs via la clause `COMPONENTS OF`, et du mode d'étiquetage par défaut `AUTOMATIC TAGS`. Ces deux concepts sont intrinsèquement liés comme le dit (ITU, 1994a, § 22.3) : la décision d'appliquer la production automatique d'étiquettes se fait *avant* l'insertion des champs, mais est appliquée *après* (ITU, 1994a, § 22.7). La décision d'appliquer ou non la production automatique d'étiquettes se fonde sur la présence d'étiquettes sur les champs, c'est pourquoi nous avons précédemment laissé en place celles-ci, alors que nous avions extrait les types des champs avec leurs contraintes de sous-typage. Illustrons le fonctionnement des clauses `AUTOMATIC TAGS` et `COMPONENTS OF`, en laissant de côté la cinquième étape qui réécrit les types internes. Par exemple,

```
SAMPLE DEFINITIONS AUTOMATIC TAGS ::=
BEGIN
```

```
A ::= SET {
        a SET OF [2] REAL,
        b [1] INTEGER (0|1)
      }
END
```

reste invariant car le champ de label `b` possède une étiquette. Cependant

```
SAMPLE DEFINITIONS AUTOMATIC TAGS ::=
BEGIN
A ::= SET {
        a SET OF [2] REAL,
        b INTEGER (0|1)
      }
END
```

est, conformément à la norme, réécrit en

```
SAMPLE DEFINITIONS AUTOMATIC TAGS ::=
BEGIN
A ::= SET {
        a [0] SET OF [2] REAL,
        b [1] INTEGER (0|1)
      }
END
```

Un exemple d'interaction avec la clause `COMPONENTS OF` est le suivant :

```
SAMPLE DEFINITIONS AUTOMATIC TAGS ::=
BEGIN
A ::= SET {
        a SET OF [2] REAL,
        COMPONENTS OF B
      }
B ::= SET {b [1] INTEGER (0|1)}
END
```

Ici, il est incorrect de remplacer l'indication `COMPONENTS OF` par le texte du champ de label `b` du type `B`, et *ensuite* de conclure que `AUTOMATIC TAGS` ne s'applique pas à cause de l'étiquette `[1]`. Il faut au contraire décider *avant* le traitement de la clause `COMPONENTS OF` si la substitution potentiellement induite par `AUTOMATIC TAGS` doit avoir lieu ou non. Ici, elle doit avoir lieu. Ensuite nous pouvons effectivement traiter `COMPONENTS OF` et appliquer juste après ce qui était prévu par `AUTOMATIC TAGS`. Ainsi notre exemple doit être réécrit en

```
SAMPLE DEFINITIONS AUTOMATIC TAGS ::=
BEGIN
A ::= SET {
```

```
        a [0] SET OF [2] REAL,
        b [1] [1] INTEGER (0|1)
      }
B ::= SET {b [1] INTEGER (0|1)}
END
```

La **huitième étape** s'attaque à la dépendance qui existe entre les valeurs et les types, autrement qu'à travers les déclarations de valeurs, qui sont toujours typées. Cette dépendance prend deux aspects : d'une part les valeurs par défaut (annotation des champs `DEFAULT`) et d'autre part les constantes entières, énumérées et de chaînes de bits. Examinons le premier cas (section 6.5.1 page 128). Par exemple :

```
A  ::= SET {i #1 DEFAULT 7}
#1 ::= INTEGER
```

Nous réécrivons en :

```
A  ::= SET {i #1 DEFAULT $1}
#1 ::= INTEGER
$1 #1 ::= 7
```

Les noms de valeurs que nous créons sont préfixés par le symbole `$`, et sont absolument uniques le temps de l'analyse. Le principe est donc de mettre au lieu des valeurs par défaut des abréviations de valeurs dont nous introduisons la déclaration. Notons que le monde est bien fait, car depuis la cinquième étape les types des champs sont des abréviations, ici `#1`, donc la nouvelle déclaration de valeur, ici `$1`, est bien découplée de son type, c'est-à-dire que son type est une abréviation. Nous profitons ainsi de toutes les réécritures précédentes sur les types et les déclarations de valeurs. Remarquons que si les valeurs par défaut sont déjà des abréviations, une nouvelle abréviation est tout de même introduite, l'optimisation consistant à ne pas le faire étant un rafinement inutile pour l'instant. Maintenant nous pouvons nous attacher à traiter les constantes entières (section 6.5.2 page 129) :

```
A ::= INTEGER {x(0), y(1)}
```

est réécrite en :

```
A ::= INTEGER {x($1), y($2)}
$1 #1 ::= 0
$2 #2 ::= 1
#1 ::= INTEGER
#2 ::= INTEGER
```

De même que

```
Tao ::= ENUMERATED {yin(0), yang(1)}
T   ::= BIT STRING {lsb(0), msb(7)}
```

est réécrit en :

```
Tao ::= ENUMERATED {yin($1), yang($2)}
$1 #1 ::= 0
$2 #2 ::= 1
#1 ::= INTEGER
#2 ::= INTEGER

T ::= BIT STRING {lsb($3), msb($4)}
$3 #3 ::= 0
$4 #4 ::= 7
#3 ::= INTEGER
#4 ::= INTEGER
```

Nous reparlerons en détail de ces constantes lors de la mise en forme des valeurs, car elles présentent des subtilités que la norme n'envisage pas.

La **neuvième étape** découple les valeurs des contraintes de sous-typage (section 6.6 page 130). En effet, les contraintes de sous-typage contiennent de nombreuses valeurs, car le paradigme ensembliste sous-tend le sous-typage d'ASN.1. Un exemple extrêmement simple est le suivant :

```
T ::= INTEGER (0|1)
```

Il est réécrit en

```
T ::= INTEGER ($1 | $2)
$1 T ::= 0
$2 T ::= 1
```

Notons que le type des valeurs ainsi extraites des contraintes est toujours une abréviation du type contraint, car la cinquième étape a fait *remonter* les contraintes au niveau des déclarations (nous avons donc toujours un nom de type à portée de la main). Les contraintes étant très complexes, cette réécriture est très longue et délicate à formaliser.

L'étape suivante met en forme les valeurs, de la même façon que nous avons mis en forme les types au préalable. Le danger qu'il y a à mettre en œuvre cette étape est la non-terminaison à cause de la possibilité des valeurs récursives. Nous allons procéder en deux temps. Tout d'abord nous allons résoudre le problème posé par l'ambigüité du lexème `0` (section 6.7.1 page 132). Ce sera la **dixième étape**. Nous pouvons écrire :

```
x INTEGER ::= 0
y REAL ::= 0
```

Cette ambigüité poserait des problèmes dans le modèle si elle n'était pas levée ici. Pour ce faire, nous étendons la syntaxe normalisée par un nouveau mot-clé, `0.0`, qui distingue le zéro de type `REAL`. On remarquera que la levée de cette ambigüité suppose un contrôle partiel des types du

module, alors que le chapitre 7 page 143 est spécifiquement consacré au contrôle de types. Nous retrouvons le problème qui charrie à travers cette thèse : la très grande dépendance mutuelle des concepts d'ASN.1, donc le manque de distinction entre les phases de l'analyse, qui explique la piètre qualité de beaucoup d'outils. Nous réécrirons donc :

```
x INTEGER ::= 0
y REAL ::= 0.0
```

La **onzième étape** clos la mise en forme des valeurs par une foule de réécritures. Citons peut-être la plus importante : le dépliage des valeurs et l'interdiction de toutes les valeurs récursives. Ainsi nous acceptons le type bien fondé :

```
T ::= SET OF T
```

mais nous refusons dorénavant

```
x T ::= {x}
y T ::= {{y}}
z T ::= {{{z}}}
```

Le dépliage a pour conséquence que les valeurs ne contiennent plus du tout d'abréviations. En effet, les types, bien que dépliés, contiennent des abréviations de types s'ils sont structurés :

```
#1 ::= SET OF #2
#2 ::= INTEGER
x #1 ::= {y, 2, y}
y #3 ::= 3
#3 ::= INTEGER
```

devient :

```
#1 ::= SET OF #2
#2 ::= INTEGER
x #1 ::= {3, 2, 3}
y #3 ::= 3
#3 ::= INTEGER
```

Le lecteur de la partie formelle associée comprendra que l'intérêt d'un tel dépliage, en interdisant les valeurs récursives, permet de s'affranchir des environnements de valeurs en contexte des règles d'inférence. Nous noterons d'autre part que l'interdiction des valeurs récursives ne figure pas encore dans la norme, mais que cette décision est en voie d'être adoptée par l'ISO sur notre proposition. Cette interdiction est essentiellement fondée sur le fait que les codages normalisés sont fondés sur une sémantique stricte, et donc ne termineraient pas en présence de valeurs récursives. Nous montrerons cependant que ces valeurs ne présentent pas de problèmes pour le contrôle de types, dans le chapitre consacré au contrôle de types. Le choix de déplier les valeurs *avant* le contrôle des types reflète un choix d'interprétation profond. En effet la spécification :

```
T1 ::= SET {a BOOLEAN, b INTEGER OPTIONAL}
T2 ::= SET {a BOOLEAN}
x T1 ::= {a TRUE}
y T2 ::= x
```

doit-elle être rejetée ou bien réécrite en :

```
T1 ::= SET {a BOOLEAN, b INTEGER OPTIONAL}
T2 ::= SET {a BOOLEAN}
x T1 ::= {a TRUE}
y T2 ::= {a TRUE}
```

Dans le premier cas, cela suppose que nous retardions le dépliage des valeurs jusqu'après le contrôle des types qui poserait la question : `T1` est-il *équivalent* (ou *compatible*) avec `T2` ? Et, selon la réponse, nous déplirions ou non la valeur `y`. Nous préférons défendre notre position consistant à ne pas attendre le contrôle des types pour déplier les valeurs. En effet, nous évitons ainsi la définition délicate d'une relation d'équivalence entre types, qui devra prendre en compte des propriétés générales (à définir) de tous les codages admissibles, sans dépendre de l'un en particulier. De plus, la formalisation en est grandement simplifiée. N'oublions pas que, d'une part, les abréviations de valeurs sont lexicalement identiques aux constantes distinguées entières et aux valeurs d'énumération, et, d'autre part, que les valeurs récursives ne peuvent être éliminées que lors du dépliage des valeurs. Ce choix de la simplicité a pour conséquence que nous acceptons plus de spécifications que dans l'autre possibilité (voir le dernier exemple). Pour être complet, notons que dans la sous-phase précédente, nous avons levé l'ambigüité du lexème `0`, donc

```
x REAL ::= 0
y INTEGER ::= x
```

aura été réécrit en

```
x REAL ::= 0.0
y INTEGER ::= x
```

puis à cette étape

```
x REAL := 0.0
y INTEGER ::= 0.0
```

ce qui sera rejeté par le contrôle des types. Nous ne réécrivons donc pas

```
x REAL ::= 0
y INTEGER ::= x
```

en

```
x REAL ::= 0
y INTEGER ::= 0
```

ce qui serait tout à fait incorrect.

Une autre remarque concerne le traitement des types `ENUMERATED` lors du dépliage des valeurs. Nous prenons le parti de traiter les valeurs de ces types de façon spéciale : nous accepterons une abréviation de valeur d'un type énuméré si et seulement si les deux types (celui de l'abréviation et celui de la valeur abrégée) sont nominalement identiques ou obtenus par sous-typage d'un même type, c'est-à-dire d'une même déclaration de type. Par exemple la spécification

```
A ::= ENUMERATED {a(0), b(1)}
B ::= A
x A ::= b
y B ::= x
```

sera acceptée, et

```
A ::= ENUMERATED {a(0), b(1)}
B ::= A (b)
x A ::= b
y B ::= x
```

aussi, mais

```
A ::= ENUMERATED {a(0), b(1)}
B ::= ENUMERATED {a(0), b(1)}
x A ::= b
y B ::= x
```

sera *rejetée* parce que `A` et `B`, bien que structurellement identiques, sont nominalement différents, c'est-à-dire qu'ils sont définis par différentes déclarations. Nous voulons croire que cette décision, en plus d'être théoriquement très simple (sinon la plus simple), est également pragmatique, c'est-à-dire qu'elle n'invaliderait pas de spécifications actuellement normalisées ou en cours d'utilisation dans l'industrie.

Toujours au sujet des énumérés, nous donnons une valeur aux constantes qui n'en déclarent pas. Par exemple

```
T ::= ENUMERATED {x($1), y, z($2)}
$1 INTEGER ::= 7
$2 INTEGER ::= 3
```

sera réécrit en :

```
T ::= ENUMERATED {x($1), y($3), z($2)}
$1 INTEGER ::= 7
$2 INTEGER ::= 3
$3 INTEGER ::= 0
```

Nous suivons en cela la règle (ITU, 1994a, § 17.3) qui veut que nous donnions un entier en comptant à partir de zéro, en augmentant de un pour chaque constante et en sautant les entiers qui seraient déjà utilisés.

Pour revenir aux types autres que `ENUMERATED`, ajoutons que ce choix du dépliage n'entraine pas de surcoût pour le codage. En effet, le coût du codage étant négligeable par rapport à celui de la transmission sur la ligne, le dépliage des valeurs, qui peut donc entrainer le codage répété d'une même valeur abrégée, est négligeable. Par exemple la réécriture de

```
x INTEGER ::= 7
y INTEGER ::= x
```

en

```
x INTEGER ::= 7
y INTEGER ::= 7
```

entraîne que les codages de `x` et `y` soient les mêmes, car on effectue alors deux fois le même traitement, mais ce doublon est tout à fait négligeable devant la vitesse de transmission. Ceci dit, il est imaginable de ne pas déplier les valeurs et de conserver les abréviations jusqu'au codage, dans le but de partager les codages des valeurs.

Une autre réécriture, toujours lors de la mise en forme des valeurs, consiste à insérer les valeurs par défaut dépliées lorsqu'aucune valeur pour le champ n'est spécifiée. Par exemple :

```
T  ::= SET {i #1 DEFAULT $1}
#1 ::= INTEGER
$1 #1 ::= 7
t T ::= {}
```

sera réécrit en :

```
T  ::= SET {i #1 DEFAULT $1}
#1 ::= INTEGER
$1 #1 ::= 7
t T ::= {7}
```

En ceci, nous accomplissons l'exigence (ITU, 1994a, § 22.9).

Une autre réécriture importante est relative aux constantes entières et énumérées. Il convient d'abord de poser le problème. Il est possible de distinguer des valeurs entières lors de la définition du type `INTEGER` :

```
T ::= INTEGER {zero(0), un(1)}
```

de façon à permettre des définitions de valeurs de la façon suivante :

```
x T ::= zero
y T ::= un
```

tout en permettant les habituelles :

```
z T ::= 0
t T ::= 1
u T ::= 7
```

La difficulté qui apparaît ici est d'ordre lexical : on ne peut distinguer en examinant seulement la syntaxe de la valeur si cette dernière est une abréviation ou bien une constante distinguée des types `INTEGER` ou `ENUMERATED`. La clause (ITU, 1994a, § 16.4) précise la portée des constantes par rapport aux abréviations à l'aide de l'exemple qui suit :

```
a INTEGER ::= 1
T1 ::= INTEGER {a(2)}
T2 ::= INTEGER {a(3), b(a)}
c T2 ::= b
d T2 ::= a
```

La définition ci-dessus de `c` est équivalente à `c T2 ::= 1` car, dans la définition de `T2`, la valeur `a` définissant la constante `b` est forcément une abréviation (c'est la règle). Et la définition de `d` est équivalente à `d T2 ::= 3`, ce qui signifie que l'interprétation en tant que constante entière est prioritaire sur l'interprétation en tant qu'abréviation (l'autre possibilité était `d T2 ::= 1`). C'est donc cette clause de la norme que nous formalisons en réécrivant la spécification ci-dessus en :

```
a #4 ::= 1
#4 ::= INTEGER

T1 ::= INTEGER {a($1)}
$1 #1 ::= 2
#1 ::= INTEGER

T2 ::= INTEGER {a($2), b($3)}
$2 #2 ::= 3
$3 #3 ::= 1
#2 ::= INTEGER
#3 ::= INTEGER

c T2 ::= 1
d T2 ::= 3
```

Mais que faut-il comprendre par :

```
x INTEGER {c(x)} ::= c
```

Si nous appliquons nos réécritures, il vient :

```
x #0 ::= c
#0 ::= INTEGER {c(x)}
```

puis :

```
x #0 ::= $1
#0 ::= INTEGER {c($1)}

$1 #1 ::= x
#1 ::= INTEGER
```

et

```
x #0 ::= x
#0 ::= INTEGER {c($1)}

$1 #1 ::= $1
#1 ::= INTEGER
```

Il est clair que `x` et `$1` sont incorrectes. Notre définition exotique ci-dessus doit donc être rejetée car récursive (à travers un type), et notre réécriture la rejette effectivement (une trace est conservée au long des dépliages pour détecter les boucles).

Lors de la même phase de mise en forme des valeurs, nous vérifions que les valeurs des types `SEQUENCE` sont bien présentes dans l'ordre des champs de leur type. Ainsi la spécification

```
T ::= SEQUENCE {a BOOLEAN, b INTEGER}
x T ::= {b 7, a TRUE}  -- Invalide
```

est incorrecte car les valeurs dans `x` correspondant aux champs de `T` doivent être dans l'ordre de ces champs (ici `a` d'abord, puis `b`). Nous effectuons cette vérification avant le contrôle des types car elle ne relève pas conceptuellement de celui-ci ni du contrôle des sous-types. En effet, le contrôleur de types se sert des labels pour mettre en correspondance les valeurs et les champs — pas de la position des premières. Nous affirmons donc l'inutilité de cette contrainte, la distinction entre `SET` et `SEQUENCE` n'étant fondamentalement pertinente que pour le codage, c'est-à-dire la sémantique, elle ne devrait se manifester qu'à ce niveau. L'usage d'un mot-clé plutôt que de l'autre deviendrait ainsi une annotation pour le codage. La norme exigeant néanmoins un tel ordonnancement (ITU, 1994a, § 22.14), nous l'avons placé ici. Ainsi s'achève la description de la onzième étape de mise en forme des spécifications ASN.1. Elle correspond globalement à la section 6.7 page 132.

La **douzième étape** est très simple : elle consiste à déplier dans les types `INTEGER`, `ENUMERATED` et `BIT STRING` les valeurs associées aux éventuelles constantes. À la huitième étape nous avions effectué l'opération quasiment inverse :

```
A ::= INTEGER {x(0), y(z)}
z INTEGER ::= 1
```

était réécrite en :

```
A ::= INTEGER {x($1), y(z)}
$1 #1 ::= 0
#1 ::= INTEGER

z #2 ::= 1
#2 ::= INTEGER
```

Nous réécrivons maintenant :

```
A ::= INTEGER {x(0), y(1)}
$1 #1 ::= 0
#1 ::= INTEGER

z #2 ::= 1
#2 ::= INTEGER
```

L'intérêt de cette réécriture est que le terme de type est maintenant indépendant des déclarations de valeurs (ici la dépendance était `z`) et surtout que les constantes qu'il déclarait implicitement (dans notre example, `0`) ont maintenant une déclaration explicite qui permettra au contrôleur de types de les inspecter, et évitera au codage d'avoir à déplier partiellement ces valeurs, et donc de posséder en contexte l'ensemble des déclarations de valeurs. Rappelons aussi que les étapes précédentes ont supprimé les définitions de valeurs récursives exotiques de la forme :

```
x INTEGER {c(x)} ::= c
```

grâce à la huitième étape. Dorénavant nous pouvons faire l'hypothèse en manipulant les types `INTEGER`, `ENUMERATED` et `BIT STRING` que leurs éventuelles constantes sont bien typées (cette assertion étant inévitablement vérifiée lors du contrôle de type, nous pouvons l'introduire en tant qu'hypothèse).

La **treizième étape** (section 6.9 page 137) porte aussi sur les constantes des types `INTEGER`, `ENUMERATED` et `BIT STRING`. Pour le type `INTEGER`, nous devons vérifier que toutes les constantes possèdent des noms différents (ITU, 1994a, § 16.6). Par exemple, après l'étape précédente, nous pouvions avoir :

```
T ::= INTEGER {x(0), x(1)}  -- Invalide
```

qui est incorrect. Pour le type `ENUMERATED`, la contrainte est la même (ITU, 1994a, § 17.2). En plus nous devons nous assurer que les entiers sont positifs ou nuls (ITU, 1994a, § 17.3). Ainsi la spécification

```
T ::= ENUMERATED {x($1)}
$1 INTEGER ::= -1    -- Invalide
```

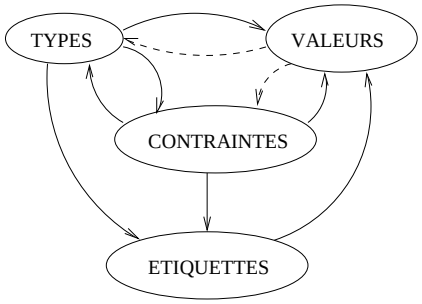

FIGURE 1.5 – Dépendances mutuelles entre concepts

sera rejetée à cette étape.

Pour le type `BIT STRING`, nous avons aussi la contrainte de positivité sur les constantes (voir (ITU, 1994a, § 19.3) et la règle de grammaire `NamedBit` (ITU, 1994a, § 19.1)) et d'unicité (ITU, 1994a, § 19.4).

La **quatorzième étape** débute une série qui clôt ce chapitre de mise en forme des spécifications ASN.1. Cette série finale (section 6.10 page 137) traite les étiquettes. En examinant la syntaxe d'ASN.1, il apparaît des dépendances supplémentaires entre types, contraintes, valeurs et maintenant étiquettes, comme on peut le voir à la figure 1.5. Les dépendances supplémentaires intrinsèques aux étiquettes sont la raison fondamentale de certains choix faits dans les étapes précédentes et de la série de réécritures débutant ici (la quatorzième étape). Il s'agit, comme depuis le début, de découpler les différents concepts les uns des autres pour les traiter plus facilement, car indépendamment. Ici, ce sont les étiquettes que nous découplons et traitons.

Rappelons que la cinquième étape avait rendus globaux les types servant à construire d'autres types. Par exemple la spécification

```
A ::= SET {
        a [0] SET OF [2] REAL,
        b [1] INTEGER (0|1)
      }
```

était réécrite en

```
A  ::= SET {
         a [0] #1,
         b [1] #2
       }
#1 ::= SET OF [2] #3
#2 ::= INTEGER (0|1)
#3 ::= REAL
```

Nous laissions alors les étiquettes des champs où elles étaient spécifiées car nous avions besoin d'elles là pour ensuite résoudre l'éventuel mode d'étiquetage par défaut `AUTOMATIC TAGS` (vois la septième étape). Nous allons maintenant hisser les étiquetages (ici `[0]`, `[1]` et `[2]`) au niveau des déclarations de types : ainsi tous les étiquetages apparaîtront à ce niveau. Pour parvenir à nos fins, nous allons nous servir de la propriété d'unicité des abréviations internes que nous avons produites à la cinquième étape. Ce qui pouvait à tort paraître un choix inefficace (en effet, on aurait pu penser laisser en place les abréviations initialement présentes) ne l'était pas : nous pouvons maintenant simplement pousser nos étiquetages internes vers les déclarations dénotées par nos abréviations internes. Notre exemple précédent devient alors :

```
A  ::= SET {
         a #1,
         b #2
       }
#1 ::= [0] SET OF #3
#2 ::= [1] INTEGER (0|1)
#3 ::= [2] REAL
```

Cette réécriture particulièrement simple est correcte car les abréviations internes (ici `#1`, `#2` et `#3`) sont uniques — donc non partagées. Pourquoi ne pas avoir déplacé les étiquetages lors de la mise en place des abréviations internes (qui abrégeaient pourtant les contraintes, c'est-à-dire que ces dernières remontaient au niveau des déclarations) ? Prenons l'exemple du module suivante :

```
SAMPLE DEFINITIONS AUTOMATIC TAGS ::=
BEGIN
A ::= SET {
        a [0] SET OF [2] REAL,
        b [1] INTEGER (0|1)
      }
END
```

Alors, en supposant que la cinquième étape déplace aussi les étiquetage vers le haut, nous aurions réécrit :

```
SAMPLE DEFINITIONS AUTOMATIC TAGS ::=
BEGIN
A  ::= SET {
         a #1,
         b #2
       }
#1 ::= [0] SET OF #3
#2 ::= [1] INTEGER (0|1)
#3 ::= [2] REAL
END
```

Mais la septième étape alors aurait réécrit cette dernière spécification en :

```
SAMPLE DEFINITIONS AUTOMATIC TAGS ::=
BEGIN
A  ::= SET {
         a [0] #1,
         b [1] #2
       }
#1 ::= [0] SET OF #3
#2 ::= [1] INTEGER (0|1)
#3 ::= [2] REAL
END
```

ce qui est erroné, car la clause `AUTOMATIC TAGS` n'aurait rien changé à la spécification initiale, et nos réécritures doivent avoir pour invariant l'application ou la non-application de cette clause.

La **quinzième étape** (section 6.10.2) consiste à introduire explicitement dans la spécification les étiquettes `UNIVERSAL` des types prédéfinis du langage, et qui ne sont jamais écrites par le spécifieur. Par exemple

```
T ::= INTEGER
```

est réécrit en

```
T ::= [UNIVERSAL 2] IMPLICIT INTEGER
```

Cette réécriture laisse invariante la sémantique de la spécification. Elle est justifiée par le fait que la définition du concept d'étiquette est floue dans la norme. D'un côté les étiquettes sont des annotations de types (*tagged types*), et d'un autre côté les types prédéfinis ont une étiquette, prédéfinie elle aussi. Nous avons décidé de séparer clairement le concept d'étiquette, donc d'étiquetage, de celui de type : nous insérons ici les étiquettes de type et nous oublions par la suite leur origine particulière.

La **seizième étape** (section 6.10.3 page 141) est consacrée au calcul des étiquetages. Nous procédons en deux temps. D'abord nous déplions les éventuelles abréviations de valeurs dans les étiquettes. Par exemple la spécification

```
T ::= [e] INTEGER
e INTEGER ::= 7
```

est réécrite en

```
T ::= [7] INTEGER
e INTEGER ::= 7
```

Au passage, nous vérifions que les valeurs d'étiquettes sont bien positives ou nulles (conformément à (ITU, 1994a, § 28.2) et à la règle de grammaire `ClassNumber` (ITU, 1994a, § 28.1)) :

```
T ::= [e] INTEGER
e INTEGER ::= -7    -- Invalide
```

est rejetée.

Par la suite, les attributs d'étiquette `IMPLICIT` et contextuels sont résolus, en fonction notamment du mode d'étiquetage par défaut du module : `EXPLICIT TAGS`, `IMPLICIT TAGS`, `AUTOMATIC TAGS` ou pas d'indication particulière. À ce point, la spécification ne possède alors plus que des étiquettes `EXPLICIT` (ITU, 1994a, § 28). Par exemple la spécification

```
EXAMPLE DEFINITIONS ::=
BEGIN
A ::= [7] INTEGER
B ::= [5] IMPLICIT [3] IMPLICIT BOOLEAN
C ::= [3] EXPLICIT REAL
END
```

est réécrite en

```
EXAMPLE DEFINITIONS ::=
BEGIN
A ::= [7] EXPLICIT INTEGER
B ::= [5] EXPLICIT BOOLEAN
C ::= [3] EXPLICIT REAL
END
```

Insistons sur le fait que cette étape n'est correcte que si l'on oublie que dans la norme les types ont des étiquettes prédéfinies, ce que nous incite à faire la quinzième étape (la précédente). Considérez le cas du type `B` ci-dessus pour mieux comprendre.

Nous vérifions au passage que l'étiquette précédant immédiatement un `CHOICE` n'est pas marquée `IMPLICIT`, suivant en cela (ITU, 1994a, § 28.8). Ainsi

```
T ::= [2] IMPLICIT CHOICE {a REAL}   --- Invalide
```

est illégale et rejetée lors de cette dernière étape.

Chapitre 2

Contrôle des types

Nous présentons ici, en termes informels, comment déterminer si une valeur canonique est d'un type canonique. La question qui résume le chapitre est : « La valeur v est-elle du type T ? ». Pour répondre à cette question, nous aurons parfois à répondre à plusieurs autres questions, parfois nous pourrons répondre immédiatement. La présentation sera faite cas par cas, selon les types dont on veut vérifier la validité pour une valeur donnée. Le principe est que si on ne peut répondre « oui » à une question à l'aide des cas suivants, alors cela implique que la réponse est négative.

2.1 Définition

Le chapitre précédent présentait comment réécrire une spécification ASN.1 de façon à la simplifier et à en vérifier un certain nombre de propriétés prévues par la norme, ou souhaitables. Cet ASN.1 simplifié était appelé noyau d'ASN.1. Nous considérons donc ici que le contrôle de type s'applique au noyau d'ASN.1, ce qui suppose que le lecteur lise le chapitre précédent.

Tout d'abord, envisageons le cas des abréviations de types. Leur traitement est simple : on remplace l'abréviation par le type qu'elle abrège et on se repose la même question. Par exemple, soit :

```
A ::= INTEGER
B ::= A
b B ::= 7
```

À la question : « La valeur 7 est-elle du type `B` ? », il faut substituer la question : « La valeur 7 est-elle du type `A` ? », puis : « La valeur 7 est-elle du type `INTEGER` ? » — question à la quelle on répondra positivement.

Le lecteur attentif se souviendra que l'exemple que nous venons de donner, même s'il donne la bonne intuition, n'est pas possible dans le noyau d'ASN.1. En effet, une des étapes de réécriture du chapitre précédent éliminait les abréviations globales, pour n'en conserver que dans les champs des types structurés, ou dans les `SET OF` et `SEQUENCE OF`. Ainsi, un exemple plus réaliste aurait été :

```
A ::= INTEGER
B ::= SET {a A}
b B ::= {a 7}
```

La question aurait été : « La valeur `7` est-elle du type `A` ? ».

Remarquons de plus qu'il n'est pas nécessaire de vérifier si les abréviations de types sont réellement des abréviations, c'est-à-dire si elles dénotent bien un type déclaré. En effet, au cours du chapitre précédent, les abréviations résultantes ont été produites de façon unique et les abréviations globales, c'est-à-dire au niveau des déclarations, ont été dépliées : toutes les abréviations du noyau d'ASN.1 sont donc valides de ce point de vue.

La cas suivant est celui du type `BOOLEAN`. Les deux seules questions dont les réponses sont affirmatives sont : « La valeur `TRUE` est-elle du type `BOOLEAN` ? » et « La valeur `FALSE` est-elle du type `BOOLEAN` ? ».

Pour le type `NULL`, la seule question dont la réponse est positive est : « La valeur `NULL` est-elle du type `NULL` ? ».

Pour le type `REAL`, plusieurs cas sont possibles. Traitons d'abord le cas des valeurs spéciales `0.0`, `MINUS-INFINITY` et `PLUS-INFINITY`. Toutes ces valeurs sont du type `REAL`. Remarquons que nous n'examinons pas la structure du type `REAL`, qui est une `SEQUENCE`, pour répondre à ces questions, car ces valeurs sont spéciales précisément parce qu'elles ne s'expriment pas comme valeurs de ce type `SEQUENCE`. Voilà la raison de les traiter ici à part.

Lors du précédent chapitre, les chaînes numériques ont toutes été réécrites sous forme de chaînes de bits. Ainsi, confrontées au type `OCTET STRING`, il suffit de s'assurer que leur longueur est un multiple de 8 pour qu'elles soient de ce type.

Pour le type `INTEGER`, tous les lexèmes représentants des entiers sont de ce type. Rappelons à ce sujet que, lors du chapitre précédent, la valeur `0` a été réservée au type `INTEGER` et que son usage pour dénoter le zéro du type `REAL` a été remplacé par celui du nouveau lexème `0.0`. D'autre part, les constantes entières, comme par exemple `zero` dans la spécification suivante :

```
A ::= INTEGER {zero(0)}
a A ::= zero
```

ont disparu car elles furent réécrites en :

```
A ::= INTEGER
a A ::= 0
```

Le cas des chaînes de caractères non numériques ne pose aucun problème, si ce n'est la vérification de l'appartenance des caractères aux différents alphabets associés. Nous n'entrerons pas dans ces détails dans cette thèse.

Pour le type `BIT STRING`, toutes les chaînes de bits sont valides.

Pour le type `ENUMERATED`, il faut s'assurer que le nom de la constante est bien présent dans la déclaration du type. Par exemple :

```
A ::= ENUMERATED {x(1)}
a A ::= x
```

À la question : « La valeur `x` est-elle du type `A` ? », il faut d'abord déplier l'abréviation `A` (premier cas de ce chapitre) et redemander : « La valeur `x` est-elle du type `ENUMERATED {x(1)}` ? ». Ensuite, il faut demander : « La constante `x` est-elle déclarée par le type `ENUMERATED {x(1)}` ? ». La réponse est « oui ».

Passons au cas des types structurés, c'est-à-dire construits à l'aide d'autres types. Tout d'abord le cas des types `SET OF` et `SEQUENCE OF`. La valeur v est une liste de valeurs. La réponse à la question de départ « La valeur v est-elle du type `SET OF` T ? » dépend de : « Chacune des valeurs contenues dans v est-elle du type T ? ». Par exemple :

```
a SET OF INTEGER ::= {0,1}
```

La question que nous nous posons d'abord est : « La valeur `{0,1}` est-elle du type `SET OF INTEGER` ? ». Nous répondrons « oui » si, et seulement si, les deux questions suivantes appellent une réponse affirmative : « La valeur `0` est-elle du type `INTEGER` ? » et « La valeur `1` est-elle du type `INTEGER` ? ». La réponse à ces deux questions est « oui », d'après le cas du type `INTEGER` vu précédemment. Il est important de noter que l'ordre dans lequel on répond à ces questions n'est pas significatif.

Envisageons maintenant le cas du type `CHOICE`. Ce cas se subdivise en deux.

1. Le premier sous-cas du cas `CHOICE` est celui où la valeur est syntaxiquement une valeur de type `CHOICE` et que le label qu'elle contient existe dans la définition du type. Par exemple :

   ```
   v CHOICE {
       a INTEGER,
       b BOOLEAN
     } ::= a : 7
   ```

   Ici, la valeur est `a : 7`, donc elle possède la syntaxe correcte pour être une valeur de type `CHOICE`. De plus, le label qu'elle contient, `a`, existe dans une des variantes du type : « `a INTEGER` ». Dans ce

cas, la question initiale qui était : « La valeur `a : 7` est-elle du type `CHOICE {a INTEGER, b BOOLEAN}` ? » devient : « La valeur `7` est-elle du type `INTEGER` ? »

2. Le second sous-cas dit qu'il est possible d'ignorer une variante, soit un champ de `CHOICE`, et donc de reposer la question initiale avec le même `CHOICE` mais avec un champ en moins. Par exemple :

   ```
   v CHOICE {
       a INTEGER,
       a INTEGER,
       b BOOLEAN
     } ::= a : 7
   ```

   La question « La valeur `a : 7` est-elle du type `CHOICE {a INTEGER, a INTEGER, b BOOLEAN}` ? » peut être équivalente à : « La valeur `a : 7` est-elle du type `CHOICE {a INTEGER, b BOOLEAN}` ? » Ce sous-cas nous permet donc d'ignorer la première variante du `CHOICE`, par exemple.

Mais pourquoi autoriser un tel type `CHOICE` qui possède deux labels identiques `a`, alors que cela est explicitement interdit par (ITU, 1994a, § 26.6) ? Tout d'abord parce que cette restriction n'est pas du tout gênante pour le contrôle des types, comme nous venons de le voir. De plus, cette propriété d'unicité des labels sera définie ci-après et mise en parallèle avec celle d'unicité des étiquetages dans le chapitre consacré à la sémantique, ce qui donnera une nouvelle perspective à la raison d'être de cette restriction.

Remarquons que nous aurions pu alors conclure que la déclaration précédente :

```
v CHOICE {
    a INTEGER,
    a INTEGER,
    b BOOLEAN
  } ::= a : 7
```

était correcte pour le contrôle des types, et ce *de plusieurs façons différentes*. La première façon consite à commencer comme ci-dessus, en éliminant la première variante :

```
v CHOICE {
    a INTEGER,
    b BOOLEAN
  } ::= a : 7
```

puis à mettre en correspondance le type de la variante `a` restante avec la valeur incluse dans `a : 7`, c'est-à-dire `7`. Ensuite la question devient : « La valeur `7` est-elle du type `INTEGER` ? », et la réponse est « oui ».

Une autre façon consiste à mettre directement en correspondance la première variante et la valeur. Lorsque nous imposerons l'unicité des labels dans les types `CHOICE`, alors nous obtiendrons l'unicité du processus de contrôle des types, c'est-à-dire que nous n'aurons alors qu'une seule façon de conclure « oui ».

Avant d'en terminer avec le cas du type `CHOICE`, notons que le contrôle des types répond « non » face à un type `CHOICE {}` qui pourait apparaître dans une question dérivée (la syntaxe l'interdit pour les types initiaux).

Traitons maintenant le cas du type `SEQUENCE`, qui se subdivise en trois.

1. Le premier sous-cas dit que la réponse à la question : « La valeur `{}` est-elle du type `SEQUENCE {}` ? » est « oui ».
2. Le second sous-cas dit que s'il existe un champ du type `SEQUENCE` dont le label apparaît dans la valeur, alors on doit répondre à deux questions : d'une part la même question mais avec ce champ et cette valeur en moins, et d'autre part la question qui met en vis-à-vis cette valeur et le type du champ de même label. Par exemple :

   ```
   v SEQUENCE {
       a INTEGER,
       b BOOLEAN
     } ::= {a 7, b TRUE}
   ```

   La question initiale est : « La valeur `{a 7, b TRUE}` est-elle du type `SEQUENCE {a INTEGER, b BOOLEAN}` ? » devient double :
   — « La valeur `b TRUE` est-elle du type `SEQUENCE {b BOOLEAN}` ? » ;
   — « La valeur `7` est-elle du type `INTEGER` ? ».
   Seule la conjonction de deux « oui » permet de répondre « oui » à la question initiale.
3. Le troisième et dernier sous-cas dit que si un champ du type `SEQUENCE` est annoté `OPTIONAL`, alors la question peut être reposée avec le même type `SEQUENCE`, mais sans ce champ. Par exemple :

   ```
   v SEQUENCE {
       a INTEGER,
       b BOOLEAN OPTIONAL
     } ::= {a 7}
   ```

   On peut alors se poser la question : « La valeur `a 7` est-elle du type `SEQUENCE {a INTEGER}` ? », puis, en appliquant le second sous-cas : « La valeur `7` est-elle du type `INTEGER` ? ». La réponse à cette dernière question, et donc à la question initiale, est « oui ».

Il est important de noter que nous n'avons pas imposé que les valeurs de la séquence soient dans l'ordre des champs du type `SEQUENCE`, comme cela

est pourtant exigé par la norme (ITU, 1994a, § 22.14). Nous considérons en effet que cela n'est pas du ressort du contrôle des types, aussi la vérification de cette exigence a été faite lors de la canonisation des types. Nous soutenons par ailleurs que cette contrainte est inutile à ce niveau, car la seule différence entre `SEQUENCE` et `SET` réside dans le codage de leurs valeurs et le contrôle sémantique des types — que l'on peut assimiler à un décodage. Par conséquent, nous traiton le cas du type `SET` comme si c'était le type `SEQUENCE`.

2.2 Unicité du contrôle des types

Comme nous l'avons dit à la section précédente, nous avons autorisé jusqu'ici les types `SET`, `SEQUENCE` et `CHOICE` dont les labels, c'est-à-dire les noms de champs, ne sont pas nécessairement distincts entre eux. Par exemple :

```
T ::= CHOICE {
        a INTEGER,
        a INTEGER,
        b BOOLEAN
      }
```

était accepté car cela ne posait aucun problème au contrôle de type : une conséquence étant que ce dernier peut éventuellement s'effectuer de plusieurs façons possibles : nous dirons que le contrôle des types est *non déterministe*. Par exemple, pour contrôler la déclaration de valeur

```
v T ::= a : 7
```

il y a deux façons : soit on met en correspondance `7` avec le premier champ de `T`, soit le second. Si nous avions eu :

```
T ::= CHOICE {
        a INTEGER,
        a NULL,
        b BOOLEAN
      }
```

alors il n'y aurait pas eu de problème non plus, car nous aurions mis en correspondance `7` et le premier champ de `T` — le second conduisant à un échec suivit alors d'un rebroussement pour envisager le premier champ. En termes opérationnels, cela implique que l'algorithme de contrôle de types doit pouvoir effectuer des rebroussements, c'est-à-dire que s'il aboutit à une impasse (une réponse négative) alors il doit rebrousser chemin et essayer d'aboutir en empruntant une autre série de questions. S'il n'y parvient pas, alors il répond définitivement par la négative.

Si nous avions eu :

```
T ::= CHOICE {
        a BIT STRING
        a NULL,
        b BOOLEAN
      }
```

alors le contrôle de type aurait échoué car aucun des champs `a` ne convenait pour typer la valeur `7`. Ce comportement est tout à fait correct.

Mais si l'algorithme de contrôle des types est non déterministe il est en revanche cohérent, c'est-à-dire qu'il ne répond pas tantôt « oui », tantôt « non » à une même question : un échec (un non) est toujours un échec, pour toutes le exécutions possibles sur la même question en entrée, et idem pour une réussite (un oui), même s'il peut aboutir à la réponse en empruntant différents chemins.

À quoi sert donc la restriction qui consiste à interdire les répétitions de labels (ITU, 1994a, § 22.10) ? Nous affirmons que la raison profonde est reliée à l'unicité des exécutions des algorithmes qui peuvent être dérivés de notre travail formel. En particulier, ici, l'unicité des labels au sein d'un même type `SET`, `SEQUENCE` ou `CHOICE` a pour conséquence que le contrôle des types ne peut s'effectuer que de façon unique. Autrement dit, s'il répond « oui », alors il n'y avait qu'une seule façon de répondre.

Nous définissons formellement à la section 7.2 la propriété pour les types d'être *bien labellisés* : par définition, un type est bien labellisé si ce n'est pas un type `SET`, ni `SEQUENCE` ni `CHOICE`, ou alors (exclusivement) si ses labels sont tous distincts entre eux. Nous prouvons alors formellement à la section 7.3 le théorème suivant :

> *Soit une valeur v et un type* T *bien labellisé. Il n'existe qu'une seule façon de vérifier que v est de type* T.

Le but fondamental de la restriction sur les doublons de labels n'est pas du tout formulé dans X.680. Nous affirmons que le but est d'assurer l'unicité des exécutions de l'algorithme de contrôle des types, et nous le prouvons formellement. Nous démontrerons formellement aussi dans le chapitre consacré à la sémantique (8 page 159), que cette même restriction a pour conséquence d'assurer l'unicité du codage des valeurs ASN.1. De même, la restriction des doublons des étiquetages des champs aura pour conséquence de garantir l'unicité du décodage.

Une façon pratique d'évaluer intérêt d'avoir l'unicité du contrôle des types, comme celle du codage et du contrôle sémantique, est que l'implémentation est alors nettement plus efficace, car il n'y a plus besoin dans ce cas de conserver une trace des questions intermédiaires, dans l'hypothèse où il faudrait rebrousser chemin.

Chapitre 3

Sémantique

Syntaxe et sémantique sont généralement apposées. La syntaxe ne suppose aucune interprétation, autrement dit elle ne fait qu'exprimer, sans dire ce qu'elle exprime. En aucun cas elle ne définit des concepts : elle ne les dénote que si elle est complétée par une sémantique. La sémantique s'exprime aussi, donc est syntaxique de ce point de vue ; c'est son apposition avec ce que l'on dénomme simplement syntaxe qui la rend « sémantique » à proprement parler. En principe la sémantique dénote des concepts de haut niveau, souvent mathématiques (c'est-à-dire que la sémantique de la sémantique est les mathématiques), donc plus abstrait que la (simple) syntaxe. Par exemple, en théorie de la programmation, la sémantique (dynamique) est les valeurs mathématiques et le mécanisme d'évaluation. (Il faut distinguer le caractère `1`, syntaxique, du nombre mathématique 1. Idem pour les opérations arithmétiques, etc.) Dans le cas d'ASN.1, nous n'avons pas d'évaluation, car nous ne spécifions pas de calculs (opérations arithmétiques, fonctions, procédures, etc.). Nous dirons que la sémantique du noyau d'ASN.1 est, au sens large, le codage des valeurs et l'ensemble des codes possibles, et au sens strict seulement l'ensemble des codes possibles. Notons qu'il peut y avoir plus de codes possibles que de valeurs codables et que ce n'est pas un problème.

3.1 Définition

Nous noterons $[\![v]\!]_{\mathrm{T}}$ le code d'une valeur v dont on suppose qu'elle est de type T. La première remarque est qu'*a priori* il n'est pas nécessaire que la valeur v soit effectivement du type T — ce que certifie le contrôle des types présenté au chapitre 2 page 37 — même si dans la pratique l'émetteur s'assurera d'abord de cette propriété avant de coder v. La seconde remarque concerne le receveur. Son univers du discours est

constitué de codes et non de valeurs. Ainsi, lorsqu'il reçoit un code c, il doit se demander, puisque par hypothèse il partage avec l'émetteur la même spécification (c'est-à-dire la définition de T), si ce code est un code possible pour une valeur du type T, autrement dit si $\exists v.[\![v]\!]_{\mathrm{T}} = c$. Une preuve constructive de cette dernière proposition serait un décodage, mais nous nous contenterons ici d'établir une propriété qui, si elle est satisfaite, implique la possibilité d'un décodage : *c'est le contrôle sémantique des types*. Cette relation ressemble à celle de contrôle des types vue au chapitre 2 page 37, sauf qu'elle confronte types et codes, et non types et valeurs.

Dans ce chapitre, nous présenterons d'abord un codage de référence. Nous employons le terme « codage » pour désigner le processus ou l'algorithme qui produit un *code* à partir de la donnée d'une *valeur* et d'un *type*. Ensuite, nous introduirons le contrôle sémantique des types. Nous pourrons alors présenter à grands traits un théorème important : *la correction sémantique du codage*. Cela signifie *grosso modo* que si, dans le noyau d'ASN.1, une valeur est bien typée, alors le code de cette valeur sera lui aussi bien typé. D'une certaine façon, cela peut être compris comme la conservation du type des valeurs par le codage de référence. Nous ajouterons à cet important résultat (qui concerne la sûreté de la transmission) deux autres concernant l'unicité des exécutions du codage et du contrôle sémantique des types : si les types du noyau d'ASN.1 possèdent des étiquetages de champs « suffisamment distincts entre eux », alors nous obtenons l'unicité du contrôle sémantique des types ; et, d'autre part, s'ils possèdent des noms de champs tous distincts entre eux, alors le codage sera unique. Ces deux derniers résultats affirment le *déterminisme* de la transmission, complétant ainsi celui de sa correction.

Note : Dans les exemples de ce chapitre, et les suivants, le mode d'étiquetage par défaut est `EXPLICIT`, à moins d'indication contraire.

3.2 Codage

Le codage de référence que nous introduisons dans notre thèse (formellement à la section 8.1 page 159) s'inspire de ceux normalisés ITU (1994e), en particulier les CER (*Canonical Encoding Rules*) et les DER (*Distinguished Encoding Rules*). Le but ici n'est pas de capturer tous les aspects d'un codage réel, mais d'en saisir les parties qui servent notre propos, par exemple les subtilités du codage des valeurs de type `REAL` ne présentent ici aucun intérêt. Le codage par les BER (*Basic Encoding Rules*) ne convient pas, même simplifié, car il ne définit pas une fonction de codage, c'est-à-dire une correspondance entre des valeurs et *exacte-*

ment un code. Les BER ne respectent pas cette condition ; par exemple le codage de la valeur `TRUE` peut se faire à l'aide de n'importe quelle valeur entière strictement positive et il n'y a donc pas unicité du code.

La syntaxe des codes est la suivante. Un code est une paire dont la première composante est une étiquette et la seconde un contenu. Un contenu est soit *primitif*, soit *construit*. Les contenus primitifs correspondent aux codes des valeurs des types `INTEGER`, `BIT STRING`, `OCTET STRING`, `BOOLEAN`, `NULL`, chaînes de caractères et aux valeurs spéciales `0.0`, `PLUS-INFINITY` et `MINUS-INFINITY` du type `REAL`. Un contenu construit est quant à lui une liste de codes. Ainsi, nous ne nous intéressons ni aux optimisations (en particulier, les valeurs non spéciales du type `REAL` sont traitées ici comme les valeurs de type `SEQUENCE`) ni aux longueurs des codes (qui ne sont utiles que pour piloter l'environnement d'exécution du décodage).

Présentons brièvement le principe de codage pour les types structurés.

Une valeur de type `SET OF` ou `SEQUENCE OF` est constituée d'un ensemble avec répétition de valeurs. Chacune d'elles est codée, sachant quel est le type de ces valeurs (éléments). Si le type est `SEQUENCE OF` alors le code possède un contenu construit à partir de ces sous-codes *dans le même ordre que dans la valeur*. Nous employons le terme « sous-code » pour désigner un code qui sert à contruire le contenu d'un autre. Soit

```
T ::= SET OF INTEGER
t T ::= {3,1,4,1,6}
```

Alors $[\![\texttt{t}]\!]_{\texttt{T}}$ est un code dont le contenu est construit par la liste des codes respectivement des valeurs `3`, `1`, `4`, `1` et `6`, qui sont toutes de type `INTEGER`. Dans le cas d'un type `SET OF` n'importe quelle permutation des sous-codes convient.

Le codage d'une valeur de type `CHOICE` est très simple. En examinant une par une les variantes, c'est-à-dire les champs du `CHOICE`, il y a deux possibilités : ou bien on laisse de côté la variante, ou bien son label est identique à celui contenu dans la valeur et dans ce cas le code de cette valeur sera le code de la valeur qu'elle contient, sachant que cette dernière valeur est du type de la variante. Considérons par exemple :

```
T ::= CHOICE {
        a INTEGER,
        a INTEGER,
        b BOOLEAN
      }

t T ::= a : 7
```

et envisageons une par une les variantes dans l'ordre d'écriture. Nous avons le choix de rejeter la première variante et de passer à la suivante,

ou de la conserver. Si nous la conservons alors $[\![\texttt{t}]\!]_{\texttt{T}} \triangleq [\![7]\!]_{\texttt{INTEGER}}$, où le symbole $\triangleq$ introduit à sa droite la définition du terme qui est à sa gauche. Sinon, la seconde variante est examinée. À nouveau, nous pouvons soit la conserver soit la rejeter. Si nous la conservons, alors $[\![\texttt{t}]\!]_{\texttt{T}} \triangleq [\![7]\!]_{\texttt{INTEGER}}$. Si nous la rejetons, nous envisageons la variante suivante. Mais nous ne pouvons conserver cette dernière car son label est `b` et diffère donc de celui contenu dans `t`, c'est-à-dire `a`. Il faut donc la rejeter, mais puisque c'était la dernière, nous devons rebrousser chemin avant de nous avouer vaincu. Nous reprenons alors la seconde étape qui examina la seconde variante et nous décidons alors de la conserver, ce qui termine l'algorithme. Notez que nous avions imposé un ordre *a priori* qui était celui de l'écriture, mais que n'importe qu'elle ordre convenait tout aussi bien. Le lecteur comprend maintenant que notre algorithme peut être amené à effectuer des rebroussements, car nous n'avons pas obligé les variantes à posséder des labels tous distincts entre eux. Le même phénomène s'était produit avec l'algorithme de contrôle des types, pour les mêmes raisons : d'un point de vue théorique, si l'algorithme est à même de rebrousser chemin, alors il n'y a pas besoin de contraindre les labels. Nous dirons que l'algorithme est non déterministe, c'est-à-dire qu'il peut répondre en empruntant différent chemins (ou raisonnements). Nous verrons plus tard qu'en contraignant les labels à être tous distincts entre eux, en accord avec X.680, alors l'algorithme devient déterministe, c'est-à-dire qu'il ne rebrousse plus jamais chemin.

Mais si l'algorithme de contrôle sémantique des types est non déterministe, il est en revanche *cohérent*, c'est-à-dire qu'il ne répond pas tantôt « oui », tantôt « non » à une même question : un échec (un « non ») est toujours un échec, pour toutes les exécutions possibles sur la même question en entrée, et idem pour une réussite (un « oui »), même s'il peut aboutir à la réponse en empruntant différents raisonnements.

Le codage d'une valeur de type `SEQUENCE` présente trois cas, alors que le type `CHOICE` en possédait deux :

1. *La valeur est `{}` et le type `SEQUENCE {}`.* Dans ce cas le contenu du code est construit avec une liste vide de sous-codes.
2. *Le type `SEQUENCE` possède un champ marqué `OPTIONAL`.* Dans ce cas il est possible d'écarter ce champ et d'envisager d'autres cas.
3. *La valeur contient une liste non vide de valeurs-éléments et le type `SEQUENCE` possède un champ dont le label égale celui d'une des valeurs listées.* Alors le contenu du code de la valeur contient le code de cette dernière valeur, dont le type est celui du champ. On recommence pour les autres champs et valeurs restantes.

De plus, les codes corespondant aux éléments du `SEQUENCE` doivent se

trouver dans le même ordre que les champs du type.

Il est très important de noter que le troisième cas recouvre le second, ce qui montre que notre algorithme est bien non déterministe. Prenons un exemple :

```
T ::= SEQUENCE {
        a INTEGER OPTIONAL,
        a INTEGER OPTIONAL
      }

t T ::= {a 7}
```

Il y a deux manières de coder `t`, toutes deux conduisant au même code (le codage est cohérent) $[\![7]\!]_{\texttt{INTEGER}}$: la première façon consiste à associer `7` et le type `INTEGER` du premier champ, et la seconde à associer `7` avec le type `INTEGER` du second champ.

Le code d'une valeur de type `SET` est le code de la même valeur comme si son type était un `SEQUENCE`, mais où les sous-codes, correspondant aux valeurs-éléments, sont permutés de façon quelconque.

3.3 Comparaison des champs

Jusqu'à présent, nous avons accepté les types `SET`, `SEQUENCE` et `CHOICE` sans restreindre les étiquetages de leur champs. La norme X.680 prévoit en fait de telles contraintes. Ainsi, pour les types `SET` et `CHOICE`, nous devrions nous assurer que tous les champs « ont des étiquetages distincts entre eux » (ITU, 1994a, § 24.3) (ITU, 1994a, § 26.2). Pour le type `SEQUENCE`, il faudrait que les champs consécutivement annotés `OPTIONAL` ou `DEFAULT`, ainsi que celui qui suit immédiatement la série, aient tous des étiquetages différents (ITU, 1994a, § 22.5). Pourquoi alors ne pas avoir tenu compte précédemment de ces clauses restrictives ? La raison est que ces restrictions n'étaient pas nécessaires pour les définitions du contrôle des types, du codage et du contrôle sémantique des types (à venir). Comme pour la restriction qui veut que les champs de ces types aient des noms tous différents entre eux, les conditions sur les étiquetages serviront à établir des propriétés d'unicité d'exécution des algorithmes associés à nos définitions formelles.

Ici nous allons commencer par examiner ce que signifie que deux champs d'un type ont des étiquetages distincts, car la norme n'est pas claire à ce sujet. Pour cela nous allons guider l'intuition à l'aide du modèle de graphe que nous avons présenté au chapitre 1 page 7 consacré au noyau d'ASN.1. Par exemple nous y avions écrit que

```
A ::= SET OF A
```

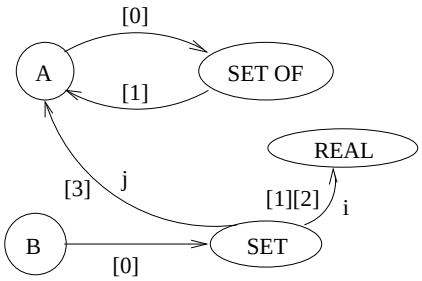

FIGURE 3.1 – Graphe avec étiquetages associé à une spécification ASN.1

```
B ::= SET {
        i REAL,
        j A
       }
```

défini le graphe de la figure 1.2 page 15.

Nous étiquetons maintenant les arcs par les étiquetages éventuels en plus des labels éventuellement déjà présents, comme ici `i` et `j`. Ainsi,

```
A ::= [0] SET OF [1] A

B ::= [0] SET {
        i [1][2] REAL,
        j [3] A
      }
```

défini le graphe de la figure 3.1.

Le lecteur attentif se souviendra que lors de la quinzième étape de réécriture du chapitre 1 page 7, nous avions introduit explicitement les étiquettes `UNIVERSAL` implicites des types prédéfinis. Par exemple :

```
T ::= INTEGER
```

était réécrit en

```
T ::= [UNIVERSAL 2] IMPLICIT INTEGER
```

Cette réécriture laisse invariante la sémantique de la spécification. Elle était justifiée par le fait que la notion d'étiquette est floue dans la norme. D'un côté, les étiquettes sont des annotations de types (*tagged types*), et d'un autre côté les types prédéfinis « ont une étiquette » (prédéfinie elle aussi). Nous avions décidé de séparer clairement le concept d'étiquette (et donc d'étiquetage) de celui de type : nous insérions ici les étiquettes de type et nous oubliions par la suite leur origine particulière. Ainsi, sur nos graphes, les annotations des arcs sont maintenant complètement

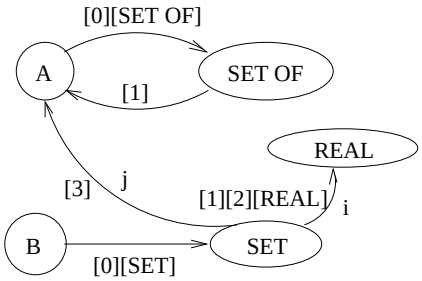

FIGURE 3.2 – Graphe avec étiquetages prédéfinis associé à une spécification ASN.1

indépendantes des sommets (voir notre exemple maintenant complété à la figure 3.2). où les étiquettes `UNIVERSAL` des types prédéfinis sont dénotées par le nom du type entre crochets, par exemple, `[INTEGER]` est mis pour `[UNIVERSAL 2]`.

Munis à présent de graphes comme modèles détaillés des spécifications ASN.1, nous allons examiner ce que disent les normes X.680 et X.690 et ce que nous formaliserons.

Intéressons-nous au type `CHOICE`, car il permet de mettre en évidence l'ensemble des difficultés. Tout d'abord rappelons l'exemple 3 de (ITU, 1994a, § 26.5) :

```
A ::= CHOICE {
        b B,
        c C
      }

B ::= CHOICE {
        d [0] NULL,
        e [1] NULL
      }

C ::= CHOICE {
        f [0] NULL,
        g [1] NULL
      }
```

Il est qualifié d'incorrect. Sa représentation graphique est donnée à la figure 3.3. Il y a problème car, à partir du sommet `CHOICE` qui suit `A`, si l'on suit les différents chemins possibles (quatre en tout) en concaténant les étiquetages des arcs, on obtient l'ensemble avec répétition {`[0][NULL]`, `[1][NULL]`, `[0][NULL]`, `[1][NULL]`}. Cet ensemble présente au moins une répétition (deux exactement), donc nous concluons que la définition

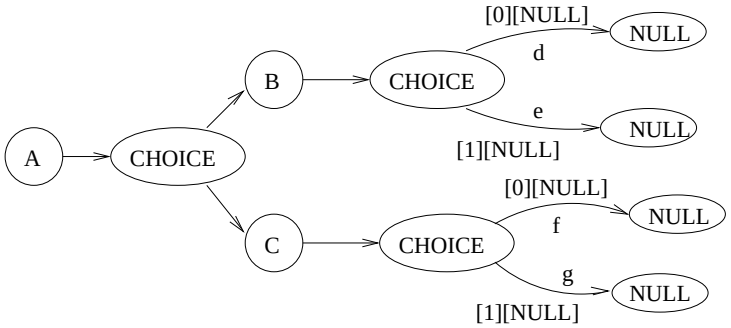

FIGURE 3.3 – Graphe associé à un exemple incorrect.

de `A` est incorrecte du point de vue des étiquetages. En d'autres termes, nous dirons que `A` est *mal étiqueté*.

Nous commençons à comprendre une première chose : ce ne sont pas les étiquetages des champs de `A` qui sont les mêmes, *c'est l'ensemble des concaténations des étiquetages des chemins à partir de `A` qui possède des doublons* (l'égalité de deux étiquetages est purement syntaxique et ne pose aucune difficulté : `[0][NULL]` est différent de `[1][NULL]`). Cette interprétation ne fait pas partie de la norme, et nous pensons qu'elle est non seulement utile pour comprendre cette dernière, mais qu'elle permet de pousser nos raisonnements plus loin encore. Ainsi, ci-dessus, l'ensemble des concaténations d'étiquetages était fini, mais il est possible qu'il soit infini, en présence de types récursifs. Par exemple :

```
T ::= CHOICE {
        a [0] T,
        b [1] NULL
      }
```

est représenté à la figure 3.4. Dans ce cas, l'ensemble des concaténations d'étiquetages à partir du sommet `T` est {`[1][NULL]`, `[0][1][NULL]`, `[0][0][1][NULL]`, ...}. Cette infinité n'est pas un problème grave car cet ensemble est un langage rationnel, c'est-à-dire que l'ensemble des étiquetage est reconnaissable par un automate fini — l'alphabet est fini puisqu'il s'agit de l'ensemble des étiquetages *de la définition du type*, qui est, par construction, de même cardinal que celui des arcs. Nous pouvons

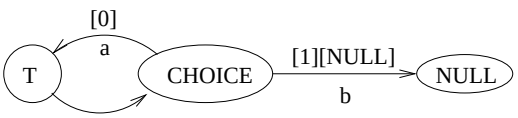

FIGURE 3.4 – Graphe associé à un type récursif.

alors construire un algorithme qui peut distinguer ou identifier les différents étiquetages de notre ensemble infini. Pour une description formelle se référer à la section 8.2.1 page 161 et 8.2.2 page 162 (formellement nous donnons un sens différent à *étiquetage* : c'est un couple formé d'une liste d'étiquettes (ce que nous nommons dans le corps de cette thèse étiquetage) et d'un type).

Remarquons que la norme X.680, malgré son flou, semble être bien plus restrictive que nous ne le sommes dans cette thèse. En effet, l'interprétation usuelle (ITU, 1994a, § 26.2) implique que :

```
T ::= CHOICE {
        a [0] INTEGER,
        b [0] NULL
      }
```

est incorrect, alors que nous l'acceptons. La raison est double.

Tout d'abord, il n'y a aucun danger à laisser passer cette déclaration car le codage de référence que nous avons défini dans cette thèse, ainsi que les codages de X.690 ITU (1994e), distinguera les codes des deux valeurs possibles du type `T`. Ainsi $[\![\texttt{a : 7}]\!]_{\texttt{INTEGER}}$ a pour étiquetage associé `[0][INTEGER]`, et $[\![\texttt{b : NULL}]\!]_{\texttt{NULL}}$ a pour étiquetage associé `[0][NULL]`. Ces deux étiquetages sont bien distincts, et il n'y a donc aucun danger de confusion pour le décodeur entre $[\![\texttt{a : 7}]\!]_{\texttt{INTEGER}}$ et $[\![\texttt{b : NULL}]\!]_{\texttt{NULL}}$.

La seconde raison est qu'en définissant comme nous l'avons fait la comparaison de champs du point de vue des étiquetages, nous pourrons réutiliser cette même comparaison lors de la définition de l'algorithme de contrôle sémantique des types, dont le succès est la condition nécessaire pour que le décodage réussisse. Du point de vue formel, nous pourrons alors réaliser plus facilement la preuve de correction du codage, car nous aurons les prémisses de comparaison de champs qui seront la même relation.

Pour aider à la compréhension des parties qui suivent, il convient de fixer le vocabulaire introduit dans cette section. Nous dirons par définition que deux champs sont *non disjoints* si leurs ensembles d'étiquetages, dont nous avons précédemment illustré le calcul, ont une intersection non vide. Par définition aussi nous dirons qu'un champ (ou variante) est *inclus* dans tel autre si son ensemble d'étiquetages est inclus dans celui de l'autre. Les lecteurs férus de théorie des langages trouveront la définition formelle de l'inclusion à la section 8.2.1 page 161, et les champs non disjoints à la section 8.2.2 page 162. Nous parlons formellement d'étiquetages de types non disjoints, car la définition formelle des étiquetages est différente de celle, plus intuitive, de cette partie informelle de notre thèse.

3.4 Contrôle sémantique des types

Le succès du contrôle sémantique des types implique celui du décodage. Il ne s'agit pas d'un véritable décodage car il est suffisant de s'assurer qu'un éventuel décodage ne poserait aucun problème. Quels pourraient être ces problèmes ? Pour cerner cette question, il faut rappeler le contexte dans lequel s'effectue la communication entre environnements *a priori* hétérogènes. Parmi les types d'un module ASN.1, ou ensemble de modules, l'émetteur en choisit un sous-ensemble que l'on dénomme en anglais *Protocol Data Unit* (PDU). Tout se passe comme si l'on définissait, à un niveau englobant la spécification ASN.1 proprement dite, un type `CHOICE` dont les variantes contiendraient ces types sélectionnés pour représenter tous les formats de données possibles au cours de la communication. Le récepteur partage avec l'émetteur. le même PDU, c'est-à-dire ce même méta-type `CHOICE`. (Nous opposons ici émetteur et récepteur, mais, bien sûr, dans la pratique l'émetteur peut devenir le récepteur et vice-versa, selon le flot de contrôle des applications communicantes.) Il sait donc *a priori* qu'il va recevoir une valeur de ce méta-type, à n'importe quel moment en général — le protocole peut être complété par une spécification temporelle des échanges, avec des synchronisations, des ordonnancements etc. Pour gagner en clarté, restreignons notre discours et imaginons que l'émetteur et le récepteur savent d'un commun accord qu'ils vont s'échanger une valeur d'un type donné à l'avance. Le récepteur ignore donc de quelle valeur il s'agit, mais connaît son type. L'émetteur vérifie d'abord que la valeur qu'il veut émettre est bien du type en question : c'est l'utilité de l'algorithme de contrôle des types que nous avons présenté informellement à la section 2. Puis il va coder cette valeur de ce type : c'est le codage de référence que nous avons esquissé informellement à la section 3.2 page 46. Ce codage possède la particularité d'inclure toutes les informations de contrôle dont nous disposons statiquement : les étiquetages, qui sont ce qui reste des types après codage et qui accompagnent les valeurs. Les valeurs sont ainsi marquées. Le récepteur doit décoder dynamiquement la valeur ainsi codée. Pour ce faire il se livre à un certain nombre de décisions en fonction des étiquetages qu'il trouve mêlés au code proprement dit, par opposition à l'information dite de contrôle pour cette raison, et que représentent les étiquetages. Ce sont ces décisions, ou vérifications, que capture notre notion de *contrôle sémantique des types*, car nous ne sommes pas intéressés par une reconstruction de la valeur initiale à partir de son code puisque ce travail est effectué dans la pratique par des fonctions automatiquement produites par le compilateur ASN.1, et elles retournent des valeurs dont le format est conforme à l'environnement d'exécution de l'application receveuse, et

jamais ne renvoient des valeurs ASN.1, qui n'existent que statiquement. Ce contrôle sémantique des types est le pendant du contrôle des types, que nous aurions pu qualifier précédemment de *syntaxique* car ce dernier s'applique aux valeurs ASN.1, alors que le premier s'appelle *sémantique* car il s'applique aux code, qui représentent la sémantique, comme nous l'avons défini au début de ce chapitre.

Présentons le principe de fonctionnement de l'algorithme de contrôle sémantique des types. Nous utiliserons pour cela les notions de champs non disjoints et de champs inclus, que nous avons définies à la section 3.3 page 49.

Tout d'abord nous définissons une petite fonction qui prend un code en argument et retourne un étiquetage. Ensuite examinons cas par cas, en fonction des types.

— Le contrôle sémantique des types `SET OF` et `SEQUENCE OF` comporte deux cas :

1. *La liste des sous-codes est vide.* On s'assure alors que l'étiquette du code est bien celle du type `SET OF` ou `SEQUENCE OF`.
2. *La liste des sous-codes est non vide.* On choisit n'importe quel sous-code et on le contrôle en le supposant du type des éléments. On recommence le traitement.

— Le contrôle sémantique du type `CHOICE` présente deux cas :

1. *La liste des variantes est non vide.* Il est possible *a priori* d'écarter n'importe quelle variante.
2. *La liste des variantes est non vide et l'étiquetage extrait du code correspond à une variante (reconstruite) incluse dans l'une de celles du type.* Dans ce cas on a reconnu le code d'une variante et on contrôle donc le code en le supposant du type de la variante.

Il est très important de noter ici que le premier cas du `CHOICE` est plus général que le second, donc que l'algorithme de contrôle sémantique des types est non déterministe, comme l'était celui de contrôle (syntaxique) des types (chapitre 2 page 37).

— Le contrôle sémantique du type `SEQUENCE` se ramène à trois cas :

1. *Le type est `SEQUENCE {}` et la liste des sous-codes est vide.* Alors on extrait l'étiquette du code et on s'assure qu'elle est bien celle du type `SEQUENCE`.
2. *La liste des champs est non vide et contient un champ `OPTIONAL`.* Il est possible d'écarter ce champ et d'examiner les autres cas.

3. *La liste des champs et la liste des sous-codes sont non vides.* On extrait du premier sous-code l'étiquetage et on vérifie que son champ reconstruit associé est inclus dans le premier des champs du type. On contrôle alors ce sous-code en supposant donc qu'il est du type de ce champ. On contrôle le reste du type par rapport au reste du code (les premiers éléments de leurs listes respectives ont été ôtés car ils viennent d'être traités.).

Il est très important de noter ici que si le champ en tête est `OPTIONAL` alors le second cas et le troisième sont également envisageables, ce qui prouve encore que l'algorithme de contrôle sémantique des types est non déterministe.

— Le cas du type `SET` est identique à celui du type `SEQUENCE`, sauf que le troisième cas correspondant ne tient pas compte de l'ordre des champs, ni de celui des sous-codes.

3.5 Correction du codage

Nous avons défini jusqu'à présent trois algorithmes :

— Le contrôle (syntaxique) des types, au chapitre 2.1 page 37 ;
— Le codage de référence, à la section 3.2 page 46) ;
— Le contrôle sémantique des types, à la section 3.4 page 54.

Comme nous l'avons esquissé à la section 3.1 page 45, nous avons démontré formellement au chapitre 8.4 page 168 le théorème dit de correction (sémantique) du codage :

> *Soit une valeur v et un type* T *du noyau d'ASN.1. Soit* $[\![v]\!]_{\mathrm{T}}$ *le code de v en supposant que v est de type* T*. Si v est de type* T*, alors* $[\![v]\!]_{\mathrm{T}}$ *est de type* T*.*

Lorsque nous disons que « v est de type T », il s'agit implicitement du succès du contrôle (syntaxique) des types, et lorsque nous disons « $[\![v]\!]_{\mathrm{T}}$ » est de type T, il s'agit implicitement du succès du contrôle sémantique des types.

On peut comprendre ce théorème comme la conservation du type des valeurs du noyau ASN.1 par le codage de référence. Autrement dit, si nous codons une valeur v de type T, et si nous tentons de la décoder en la supposant de type T, alors nous réussirons (n'oublions pas que, par construction, le succès du contrôle sémantique des types implique celui du décodage).

3.6 Unicité du codage

Nous avons montré à la section 2.1 page 37 que le contrôle des types n'était pas un algorithme déterministe. Idem pour le codage à la section 3.2 page 46 et le contrôle sémantique des types à la section 3.4 page 54. Cela signifie que l'exécution de ces algorithmes suppose la possibilité de rebroussement : si l'algorithme aboutit à une impasse alors le mécanisme d'exécution doit être à même de rebrousser chemin jusqu'à trouver un chemin de calcul qui réussisse, ou qui échoue si tous les chemins ont été essayés.

À la section 7.3 page 149 nous avons montré alors qu'en contraignant les champs des types structurés `SET`, `SEQUENCE` et `CHOICE` à posséder des labels tous distincts entre eux, alors cela impliquait l'unicité de l'exécution de l'algorithme de contrôle des types. En d'autres termes, il n'y a alors jamais plus besoin de rebrousser chemin, quelque soit l'exécution : si l'algorithme échoue alors c'est qu'il n'y a pas d'autre possibilité. L'intérêt d'une telle propriété est essentiellement de l'ordre de la complexité dynamique, c'est-à-dire de l'efficacité en temps et en espace, et de la simplicité de la programmation.

Cette contrainte sur les labels est prévue dans X.680 (ITU, 1994a), mais aucune justification n'en est donnée. Notre thèse est que sa raison d'être et son utilité est d'assurer le déterminisme du contrôle des types et du codage, c'est-à-dire l'unicité des exécutions de ceux-ci, sachant qu'ils sont cohérents, c'est-à-dire qu'ils répondent toujours de la même façon aux mêmes questions. De même, la même contrainte sur les labels implique aussi que le codage devient déterministe. Cela se comprend si l'on prête attention à la structure de sa définition par cas, qui est isomorphe à celle du contrôle des types.

3.7 Types bien étiquetés

Dans le but de parvenir au déterminisme de l'algorithme de contrôle sémantique des types, c'est-à-dire à l'unicité des exécutions de celui-ci, nous allons contraindre les types, de façon analogue à la façon dont nous avions restreint les types à être bien labellisés pour obtenir l'unicité du contrôle (syntaxique) des types et l'uncité du codage.

Nous définissons formellement à la section 8.6 page 197 un prédicat sur les types que nous nommons *types bien étiquetés*. Trois cas sont possibles :

1. T *n'est ni* `SET`*, ni* `SEQUENCE` *ni* `CHOICE`*.* Alors, T est bien étiqueté.
2. T *est* `CHOICE` *ou* `SET`*.* Si les ensembles d'étiquetages de ses champs

(présentés à la section 3.3 page 49) sont tous disjoints entre eux, alors T est bien étiqueté.

3. T *est* ***SEQUENCE.*** Si pour tous les champs d'une succession de champs marqués `OPTIONAL` ou `DEFAULT` et celui qui la suit immédiatement, leurs ensembles d'étiquetages sont tous disjoints entre eux, alors T est bien étiqueté.

Dans tous les autres cas, T n'est pas bien étiqueté.

Cette relation restreint le nombre de types valides et correspond à une généralisation des restrictions pour le type `SEQUENCE` (ITU, 1994a, § 22.5), pour le type `SET` (ITU, 1994a, § 24.3) et pour le type `CHOICE` (ITU, 1994a, § 26.4). Pourquoi une généralisation ? Parce que l'interprétation stricte de la norme X.680 nous amène à rejeter

```
T ::= CHOICE {
        a [0] [1] INTEGER,
        b [0] BOOLEAN
      }
```

En effet, la norme ne tient compte pour la comparaison des champs que de la première étiquette `[0]` de chaque étiquetage. Notre thèse est qu'une analyse récursive des étiquetages, telle que nous l'avons donnée à la section 3.3 page 49, donc plus fine, ne rend pas pour autant légales des spécifications qui poseraient des problèmes sémantiques, plus précisément : pour le contrôle sémantique des types (qui, rappelons-le, capture l'essence d'un éventuel décodage). Autrement dit, les valeurs du type `T` syntaxiquement différentes auront des codes syntaxiquement différents, car les étiquetages suffiront toujours à les distinguer, autant que la syntaxe, via les labels, permettait de distinguer les valeurs.

3.8 Unicité du contrôle sémantique des types

Nous avons montré à la section 3.4 page 54 que le contrôle sémantique des types était non déterministe, c'est-à-dire que que l'exécution de cet algorithme suppose la possibilité de rebroussement : si l'algorithme aboutit à une impasse alors le mécanisme d'exécution doit être à même de rebrousser chemin jusqu'à trouver un chemin de calcul qui réussisse, ou qui échoue si tous les chemins ont été essayés.

Comme nous l'avons déjà dit à la section 3.1 page 45, le succès du contrôle sémantique des types implique le succès d'un éventuel décodage. L'intention est donc que les propriétés du contrôle sémantique des types impliquent, par construction, une propriété semblable sur un éventuel décodage. En particulier, nous démontrons formellement à la section 8.7 page 198 le théorème suivant :

Soit un code c et un type T *bien étiqueté. Il n'existe qu'une seule façon de vérifier si c est de type* T.

Ce théorème pourrait être aussi intitulé « Déterminisme de l'algorithme de contrôle sémantique des types », et a pour conséquence pratique de rendre l'implémentation du décodage nettement plus efficace, car il n'y a alors plus besoin de conserver une trace des chemins de calculs empruntés.

Lorsque ce théorème est d'autre part mis en perspective avec les théorèmes de correction sémantique du codage (à la section 3.5 page 56), de déterminisme du contrôle syntaxique des types (à la section 2.2 page 42) ainsi que de déterminisme du codage (à la section 3.6 page 57), nous avons maintenant en plus l'assurance que *le décodage produira une valeur identique*. (N'oublions pas qu'en hypothèse de tous nos théorèmes, nous nous plaçons dans le noyau d'ASN.1, donc en particulier les valeurs sont en forme normale et ne contiennent plus d'identificateurs.)

3.9 Correction de la réécriture vers le noyau

Les théorèmes que nous avons établis s'appliquent au noyau d'ASN.1 et non pas à tout ASN.1. La raison est que la complexité du langage étant telle, la possibilité de la preuve de nos théorèmes semblait illusoire (très grande combinatoire de cas), et qu'il valait donc mieux envisager de réduire dans un premier temps les spécifications initiales vers un sous-ensemble d'ASN.1 plus simple, que nous avons nommé *noyau*. Ainsi nous ne pouvons établir formellement la correction sémantique de cette réécriture, c'est-à-dire qu'elle laisse invariants les codes des valeurs des types réécrits (ainsi que les codes des valeurs spécifiées), et étendre alors la portée de nos théorèmes à tout le langage. Néanmoins, nous pouvons nous convaincre informellement du bien fondé de cette propriété simplement en examinant étape par étape notre réécriture, et en la confrontant aux règles de codages qui sous-tendent notre codage de référence BER (ITU, 1994e). L'idée est de montrer que chaque étape laisse invariants les codes, donc leur composition aussi. Voyons étape par étape ce qu'il en est.

1. L'ajout d'une définition du type `REAL` n'influe en rien sur les codes.
2. La vérification de la bonne fondation des types ne constitue pas une étape de réécriture à proprement parlé, mais de vérification pure, donc ne change pas la forme des types, donc pas le codage de leurs valeurs (la bonne fondation assure néanmoins la terminaison du codage, bien que nous ne l'ayons pas prouvé formellement).
3. Lors du découplage entre les valeurs globales et leur types, nous

créons des abréviations de types qui sont transparentes pour le codage (règle d'inférence numéro [2] 159).

4. Lors du découplage entre les contraintes de sous-typage et les types, nous créons des abréviations de types qui sont transparentes pour le codage (règle d'inférence numéro [2] 159).
5. La réécriture des types structurés de telle sorte que ces derniers ne contiennent plus que des abréviations de type non contraintes créé des abréviations, et donc n'influe pas sur le codage (règle d'inférence numéro [2] 159).
6. Le dépliage des abréviations de type et des sélections globales n'influe en rien sur le codage (comme l'opération inverse que constitue l'abréviation).
7. Le traitement des clauses `COMPONENTS OF` et du mode d'étiquetage par défaut ajoute les étiquettes qui doivent l'être. Le codage tel qu'il est défini dans X.690 (ITU, 1994e) prévoit cette tâche : nous n'avons fait qu'avancer au niveau syntaxique cette opération sémantique.
8. Le découplage entre valeurs et types peut créer des abréviations de valeurs, qui sont sans conséquence sur le codage (comme celle des types).
9. Le découplage entre valeurs et contraintes de sous-typage peut créer des abréviations de valeurs, qui sont sans conséquence sur le codage (comme celle des types).
10. La résolution de l'ambigüité du lexème `0` (tantôt de type `INTEGER`, tantôt de type `REAL`, selon le contexte) ne fait que simplifier la tâche au codage qui aurait dû sinon la faire.
11. Le dépliage des valeurs est sans conséquence sur le codage. L'interdiction des valeurs récursives est purement de la vérification, et non une réécriture, et n'influe donc pas sur les codes.
12. Le dépliage des constantes dans les types `ENUMERATED`, `INTEGER` et `BIT STRING` est sans effet sur les codes.
13. L'unicité des constantes définies par le type `INTEGER` est de la vérification, et non de la réécriture, donc implique l'invariance des codes.
14. Le fait de remonter les étiquetages des champs des types structurés au niveau des déclarations laisse invariant les codes des valeurs de ces types structurés car les champs sont des abréviations. D'autre part, les codes de ces abréviations changent *mais* puisque ces abréviations ont été créés lors d'une étape précédente, elles ne font pas partie du PDU, c'est-à-dire qu'aucune valeur des

types qu'elles dénotent ne sera codée et envoyée directement. Par exemple, soit :

```
A  ::= SET {
         a [0] #1,
         b [1] #2
       }
#1 ::= SET OF [2] #3
#2 ::= INTEGER (0|1)
#3 ::= REAL
```

Les abréviations `#1`, `#2` et `#3` sont *par construction* uniquement accessibles à travers le type `A`. Elles ne dénotent pas des types du PDU, alors que `A` oui. Autrement dit, seul le type `A` est exporté par le module et aucun des usagers du protocole ne s'attend à échanger directement des valeurs des types `#1`, `#2` et `#3`. Cette étape de réécriture donne :

```
A  ::= SET {
         a #1,
         b #2
       }
#1 ::= [0] SET OF #3
#2 ::= [1] INTEGER (0|1)
#3 ::= [2] REAL
```

Les codes des valeurs du type `A` ne changent pas, contrairement à ceux des valeurs des types `#1`, `#2` et `#3`. Mais puisque ces derniers ne sont pas dans le PDU, cela ne change pas le protocole de communication.

15. L'introduction explicite des étiquettes `UNIVERSAL` des types prédéfinis ne fait que devancer ce que fera le codage.
16. La réduction des étiquetages ne fait que devancer ce que fera le codage.

En conclusion, la phase de réduction vers le noyau laisse invariants les codes des types de la spécification, ainsi que ceux des valeurs qu'elle contient.

Chapitre 4

Contrôle des sous-types

QUAND LE SPÉCIFIEUR veut réduire l'ensemble des valeurs d'un type, il contraint celui-ci, et forme alors un *sous-type*. Ce chapitre présente de façon informelle comment vérifier que la donnée d'un type et d'une contrainte de sous-typage est valide. La contrepartie formelle de ce chapitre se trouve en annexe à la section 9 page 217. Nous ne traiterons pas les cas des contraintes `EXCEPT` et `ALL`. Pour être très précis, nous étendrons la syntaxe du noyau d'ASN.1 par l'emploi généralisé des parenthèses. Par exemple nous préfèrerons écrire

```
T ::= SET OF (VisibleString (SIZE (5)))
```

au lieu de

```
T ::= SET OF VisibleString (SIZE (5))
```

Nous pourrons aussi écrire

```
T ::= (SET OF VisibleString) (SIZE (5))
```

au lieu de

```
T ::= SET (SIZE (5)) OF VisibleString
```

Nous pourrons aussi écrire

```
T ::= (INTEGER) ((0) | (1))
```

au lieu de

```
T ::= INTEGER (0 | 1)
```

même si cela ne présente *a priori* aucun intérêt.

Quel est le but fondamental du contrôle des sous-types ? Le contrôle des types avait ignoré les contraintes de sous-typage et avait dit si la valeur qu'on lui donnait appartenait bien au type donné. Le contrôle des sous-types s'assure qu'un sous-type n'est pas vide, c'est-à-dire qu'il

contient au moins une valeur. Ces deux aspects sont intimement liés et le parti pris de cette thèse est de séparer le contrôle des types du contrôle des sous-types pour simplifier la présentation (en particulier, le codage de référence que nous avons formalisé (inspiré des BER) ne tenant pas compte des contraintes de sous-typage, les preuves figurant en annexe auraient été énormément compliquées si contrôle des types et des sous-types n'avaient fait qu'un), mais les deux sont nécessaires pour valider une spécification. Par exemple, la déclaration de la valeur `y` dans la spécification

```
x INTEGER ::= 0
y INTEGER (0|1) ::= x
```

est analysée en deux temps :

1. D'abord on vérifie que

   ```
   y INTEGER ::= x
   ```

   est valide. La particularité ici est que les contraintes de sous-typage (éventuellement profondes dans un type structuré comme `SET` par exemple) sont ignorées. C'est le contrôle des types, présenté au chapitre 2 page 37.

2. Ensuite on vérifie que le sous-type

   ```
   INTEGER (0|1|x)
   ```

   est valide. C'est le contrôle des sous-types, qui fait l'objet de ce chapitre.

L'intérêt d'un tel découpage est la simplicité, même s'il impose alors des vérifications en deux temps. En particulier n'oublions pas que le contrôle des sous-types doit être suivi d'une nouvelle analyse pour détecter les types non fondés. Reprenons l'exemple que nous donnions dans le chapitre 1 page 7 :

```
T ::= CHOICE {a [0] T, b NULL} (WITH COMPONENTS {b ABSENT})
```

Ici intervient une contrainte de sous-typage définie (ITU, 1994a, § 45.8) qui implique que cette déclaration est équivalente à

```
T ::= CHOICE {a [0] T}
```

et nous avons déjà conclu que ce type n'était pas bien fondé. Pour formaliser cette équivalence nous procéderons à une réécriture des contraintes de sous-typage qui fait l'objet d'une phase à part entière, détaillée plus loin. Dans tous les cas elle intervient *après* la réécriture des modules ASN.1 vers le noyau d'ASN.1. Nous avons ignoré donc la contrainte de sous-typage et conclu que le type `CHOICE {a [0] T, b NULL}` était bien fondé (ce qui est absolument vrai), *mais*, après la réécriture des

contraintes de sous-typage à venir, nous devrons à nouveau vérifier que les types réécrits sont bien fondés. Ainsi nous pourrons finalement rejeter cette déclaration du type `T` (qui a passé le contrôle des types sans problèmes, car, nous l'avons dit, les contraintes y sont ignorées). Cette vérification des types bien fondés à double temps semble bien compliquée, mais la réécriture des contraintes de sous-typage est extrêmenent complexe et il est de loin préférable de procéder ainsi, plutôt que de chercher à définir une seule phase englobant tout. Pour résumer : nous avons accepté cette déclaration ci-dessus, en attendant la phase de réécriture des contraintes de sous-typage après laquelle nous réappliquons notre critère de bonne fondation.

Dans ce chapitre encore nous allons procéder par réécritures successives. Cette présentation algorithmique définit le sens des constructions sur lesquelles elle s'applique. Pour rendre ces transformations intelligibles, nous resterons au niveau de la syntaxe du noyau d'ASN.1. (Les télécommunicants entendront ici *abstraite* et les théoriciens de l'informatique *concrète*.) L'idée qui nous guidera est la réduction des contraintes complexes vers des contraintes plus simples, élémentaires. Nous devrons néanmoins introduire, pour la présentation, de nouvelles constructions qui n'ont pas d'équivalents dans la syntaxe du noyau d'ASN.1.

La **première étape** consiste à réduire les contraintes `INCLUDES`, partout où elles apparaissent : ce par quoi et à quelles conditions on les remplace définit la sémantique de ces contraintes (cette première étape correspond à la section 9.1 page 217.

Lors de la deuxième étape de réécriture du chapitre 1 page 7, nous avions accepté des déclarations telles que :

```
T ::= INTEGER (INCLUDES T)
```

bien qu'elles ne définissaient pas un type unique (ici, tout sous-type de `T` conviendrait). C'est ici que nous allons éliminer ces déclarations. En effet, lors de la quatrième étape de la réécriture du chapitre 1 page 7, nous avions produit :

```
T  ::= INTEGER (INCLUDES #1)
#1 ::= T
```

Puis, à la sixième étape :

```
T  ::= INTEGER (INCLUDES #1)
#1 ::= INTEGER (INCLUDES #1)
```

Ici, nous commençons par nous assurer que l'abréviation de la clause `INCLUDES` *ne désigne pas* le type analysé. Pour la première déclaration, il n'y a pas de problème : `#1` est différent de `T`, mais, pour la seconde, il apparaît que `#1` égale `#1`, donc elle doit être rejetée.

La deuxième et dernière vérification concernant les clause `INCLUDES` est qu'il faut s'assurer que le type inclus et celui qui l'inclut possèdent bien un sur-type en commun, c'est-à-dire qu'ils sont dérivés tous deux à partir d'un même type. Mais que signifie exactement « même type » ? La norme (ITU, 1994a, § 45.3.2) ne dit rien de plus précis, donc nous proposons ici une solution qui nous est propre : *nous dirons que deux types sont les mêmes s'ils ont un alias en commun.* Les alias nous servent à conserver une trace des abréviations de types modulo sous-typage, après les dépliages en cascade : ce sont les noms de ces abréviations. Prenons l'exemple :

```
A ::= B (0|1)
B ::= C
C ::= INTEGER
```

La phase de canonisation des environnements de types déplia les abréviations de type ci-dessus :

```
A ::= INTEGER (0|1)
B ::= INTEGER
C ::= INTEGER
```

Il nous fallait conserver trace de ces dépliages pour la phase présente, et cela grâce aux alias. Ainsi les alias de `A` sont `B` et `C`, car nous avons déplié d'abord `B`, puis finalement `C`, celui de `B` est `C`, car nous avons déplié `C`, et `C` n'en a pas, car ce n'était pas une abréviation, par définition.

Comment utilise-t-on les alias dans le cas qui nous occupe ? Considérons l'exemple :

```
Day ::= ENUMERATED {monday (0), tuesday (1), wednesday (2),
                    thursday (3), friday (4), saturday (5),
                    sunday (6)}

Week-End ::= Day (saturday | sunday)

Long-Week-End ::= Day (INCLUDES Week-End | monday)
```

Pour valider ici l'emploi de la clause `INCLUDES`, nous devons nous assurer que les types `Long-Week-End` et `Week-End` ont un sur-type commun. Pour en décider nous devons comparer leurs alias et vérifier qu'il y a au moins un alias en commun. Pour ne pas encombrer la présentation nous ne déplierons pas les types explicitement, mais il est clair que l'unique alias du type `Long-Week-End` est `Day` et *idem* en ce qui concerne `Week-End`. Donc cette spécification est valide et la définition de `Long-Week-End` est enfin réécrite en

```
Long-Week-End ::= Day ((saturday | sunday) | monday)
```

Nous avons pris les contraintes du type `Week-End` et les avons substituées (textuellement) à la clause `INCLUDES`, en prenant soin tout de même de placer des parenthèses autour du terme substitué. Ces parenthèses ne sont pas permises *a priori* par la grammaire du noyau d'ASN.1, aussi nous les y ajoutons. Une réécriture simplificatrice supplémentaire produirait (en fait elle intervient à l'étape qui suit) :

```
Long-Week-End ::= Day (saturday | sunday | monday)
```

La **seconde étape** du contrôle des sous-types consiste à mettre les contraintes en *forme disjonctive* (pour la description formelle, voir la section 9.2 page 218). Pour bien comprendre ce que cela signifie, nous devons voir les contraintes comme des expressions ensemblistes bien formées (leur syntaxe est correcte) et constituées de parenthèses, de contraintes et de deux types d'opérateurs : l'union de contraintes (`|`), ainsi que l'intersection de contraintes (`^`). Ce dernier opérateur est souvent implicite dans la grammaire d'ASN.1, où il est dénoté par la juxtaposition :

```
T ::= INTEGER (0)(1)
```

est une simplification de

```
T ::= INTEGER ((0) ^ (1))
```

et en aucun cas de

```
T ::= (INTEGER (0))(1)  -- Incorrect
```

car un sous-type est une paire dont le premier composant est un type et le second une contrainte (notre dernier exemple signifierait qu'un sous-type est une paire constituée d'un sous-type et d'une contrainte).

Toujours au sujet de l'opérateur d'intersection de contraintes (`^`), il est important de noter qu'il est commutatif, malgré les bornes spéciales `MIN` et `MAX`. En effet :

```
T ::= INTEGER (0..5)(MIN..3)
```

a le même sens que

```
T ::= INTEGER (MIN..3)(0..5)
```

même si, dans le premier cas, `MIN` a pour sémantique l'entier mathématique 0 et dans le second $-\infty$. Les bornes `MIN` et `MAX` seront remplacées respectivement par la valeur qu'elles dénotent (nous laisserons en place `MIN` et `MAX` lorsqu'elles dénoteront des infinis).

Nous dirons qu'une contrainte de sous-typage est en *forme disjonctive* si elle s'exprime comme une union de contraintes qui ne contiennent pas d'unions (pour la description formelle se référer à la section 9.2 page 218). En arithmétique on parlerait plus couramment de *forme complètement développée*, dans le cas d'une expression arithmétique simple (avec des entiers, des parenthèses et les opérateurs $+$ et $-$). La grammaire d'ASN.1 permet parfois de spécifier des formes disjonctives. Par exemple, dans la spécification

```
A ::= INTEGER
B ::= INTEGER (0)
C ::= VisibleString (SIZE (2 | 3))
D ::= VisibleString (SIZE (2..3))
```

les contraintes des sous-types `A` et `B` sont trivialement en forme disjonctive. Celle du sous-type `C` n'est pas en forme disjonctive car elle peut être réécrite en

```
C ::= VisibleString (SIZE (2) | SIZE (3))
```

ou bien :

```
C ::= VisibleString (SIZE (3) | SIZE (2))
```

car l'opérateur `|` est commutatif, et donc il n'y a pas unicité des formes disjonctives.

La contrainte du sous-type `D`, elle, est en forme disjonctive, car nous ne cherchons pas à transformer les contraintes d'intervalles en unions de contraintes de valeurs (ce n'est pas possible en général, car un intervalle peut être infini). Considérons encore d'autres exemples :

```
E ::= VisibleString (FROM ("0".."9") | SIZE (4)) (SIZE (2))
```

Ici, cette spécification serait réécrite avec une contrainte disjonctive de la façon suivante :

```
E ::= VisibleString ((FROM ("0".."9")) ^ (SIZE (2))
                     | (SIZE (4)) ^ (SIZE (2)))
```

La spécification

```
Morse ::= VisibleString (FROM ("-" | "." | " "))
```

reste invariante car elle est déjà en forme disjonctive.

Après avoir mis en forme disjonctive les contraintes du module analysé, nous distribuons les contraintes par rapport à leur type (formellement, à la section 9.3 page 219) : ainsi les contraintes ne s'expriment plus à l'aide de l'opérateur `|`, et sont donc plus simples. En échange de cette simplicité des contraintes, nous devons introduire et faire usage d'un nouvel opérateur : l'union de sous-types (notée $\cup$). Par exemple la définition

```
T ::= INTEGER (0|1)
```

est réécrite en

```
T ::= (INTEGER (0)) ∪ (INTEGER (1))
```

ou, plus simplement :

```
T ::= INTEGER (0) ∪ INTEGER (1)
```

La sémantique de cet opérateur est l'union des ensembles de valeurs correspondant aux deux sous-types sur lesquels il s'applique. Cette distribution des formes disjonctives par rapport à leur type permet de faire apparaître des sous-types très simples, que l'on peut traiter séparément (ici : `INTEGER (0)` et `INTEGER (1)`).

Donnons quelques exemples de distribution de formes disjonctives :

```
T ::= SET (WITH COMPONENT (0) | WITH COMPONENT (1)) OF INTEGER
```

devient :

```
T ::= SET (WITH COMPONENT (0)) OF INTEGER
    ∪ SET (WITH COMPONENT (1)) OF INTEGER
```

Et

```
T ::= SET {a INTEGER, b BOOLEAN}
      (  WITH COMPONENTS {a (0), b (TRUE)}
       | WITH COMPONENTS {a (1), b (FALSE)}
      )
```

devient :

```
T ::= SET {a INTEGER, b BOOLEAN}
          (WITH COMPONENTS {a (0), b (TRUE)})
    ∪ SET {a INTEGER, b BOOLEAN}
          (WITH COMPONENTS {a (1), b (FALSE)})
```

Ce dernier exemple est particulièrement intéressant. En effet, il nous servira à montrer que les calculs simplificateurs sur les sous-types peuvent conduire à écrire des définitions qui ne peuvent être exprimées simplement à partir de la grammaire d'ASN.1. En anticipant un peu, nous réduirons cette précédente forme en :

```
T ::= SET {a INTEGER (0), b BOOLEAN (TRUE)}
    ∪ SET {a INTEGER (1), b BOOLEAN (FALSE)}
```

et il est clair qu'il est impossible d'écrire sans contraintes un type (à l'aide de la grammaire d'ASN.1) qui ait exactement pour valeurs celle du type `T` ainsi réécrit (ses valeurs sont : `{a 0, b TRUE}` et `{a 1, b FALSE}`). Insistons bien sur le fait que nos réécritures ne rajoutent pas des types précédemment impossibles, mais qu'il n'est pas toujours possible d'écrire un type sans contraintes qui contienne exactement les mêmes valeurs qu'un autre sous-type donné. Nos réécritures laissent invariants les ensembles de valeurs des sous-types auxquelles elles s'appliquent. Nous reviendrons plus tard sur ce point délicat. Retenons pour l'instant que le

nouvel opérateur ∪ est parfois nécessaire pour exprimer très simplement des sous-types initialement très complexes.

Une remarque au sujet de l'emploi de l'opérateur d'union de sous-types (∪) : nous ne formons pas et ne chercherons pas à former des unions de sous-types disjointes. Par exemple la contrainte de `VisibleString` dans

```
T ::= VisibleString (SIZE (1..2 | 2..3))
```

fut d'abord mise en forme disjonctive :

```
T ::= VisibleString (SIZE (1..2) | SIZE (2..3))
```

puis a été réécrite en

```
T ::= (VisibleString (SIZE (1..2)))
    ∪ (VisibleString (SIZE (2..3)))
```

et il est clair que les chaînes de caractères de longueur égale à 2 sont des valeurs pour les deux sous-types dont on forme ici l'union.

Pour conclure cette seconde étape de réécriture pour le contrôle des sous-types, rappelons que depuis la cinquième étape de la mise en forme canonique (au chapitre 1 page 7) les contraintes n'apparaissent plus dans les types structurés — donc cette présente étape de réécriture ne s'effectue pas récursivement sur les sous-types.

La **troisième étape** consiste à recoupler les valeurs et les contraintes (la contrepartie formelle se trouve à la section 9.4 page 219). Lors de la phase liminaire de mise en forme canonique des modules ASN.1 (neuvième étape du chapitre 1 page 7), c'est-à-dire la réécriture vers le noyau d'ASN.1, nous avions découplé (i.e. séparé) les valeurs et les contraintes. Cela revient à dire que les valeurs qui apparaissaient dans les contraintes étaient remplacées par une abréviation. Par exemple :

```
T ::= INTEGER (0|1)
```

était réécrit en :

```
T ::= INTEGER ($1 | $2)
$1 T ::= 0
$2 T ::= 1
```

Nous allons maintenant déplier les abréviations que nous avions créées :

```
T ::= INTEGER (0|1)
$1 T ::= 0
$2 T ::= 1
```

De cette façon, le traitement des contraintes pourra ignorer l'ensemble des définitions de valeurs.

La **quatrième étape** a pour but de réduire les contraintes `WITH COMPONENT` et `WITH COMPONENTS` (que nous appellerons *contraintes internes*, car elle s'appliquent à des types qui se trouvent dans d'autres types — dits, pour cette raison, *structurés*), ainsi que les contraintes par valeur qui peuvent s'y ramener (la contrepartie formelle de cette étape se trouve en annexe 9.5 page 220). Les contraintes internes sont expliquées dans (ITU, 1994a, § 45.8). Mais commençons par une remarque : la norme X.680 ne dit pas ce qu'il faut penser de

```
A ::= SET {x INTEGER}
B ::= SET {COMPONENTS OF A} (WITH COMPONENTS {x (0|1)})
```

Dans cette thèse nous avons décidé, lors de la septième étape de la mise sous forme canonique, de remplacer les clauses `COMPONENTS OF` par les champs qu'elle désigne. Par conséquent nous réécrivons d'abord la spécification ci-dessus en :

```
A ::= SET {x INTEGER}
B ::= SET {x INTEGER} (WITH COMPONENTS {x (0|1)})
```

Comme nous le verrons plus loin, cette dernière spécification est valide, donc la première aussi. (Cette réécriture n'ayant pas d'équivalent au niveau du codage, nous ne pouvons parler de correction de celle-ci. Il est donc admis que cette réécriture laisse invariante la sémantique.) Avant d'entrer dans le vif de cette quatrième étape, commençons par énoncer quelles remarques qui serviront de fil d'Ariane.

Tout d'abord, quel intérêt y a-t-il à éliminer les contraintes internes ? Le premier avantage est que nous faisons ainsi apparaître des contraintes plus élémentaires. Le second avantage est lié a la façon dont les contraintes internes s'appliquent aux types. Soit par exemple

```
T ::= SET {a INTEGER (0|1)} (WITH COMPONENTS {a (16)})
```

Pour pouvoir rejeter cette déclaration il nous faut confronter les contraintes `(0|1)` et `(16)`, donc décomposer le type `SET` au fur et à mesure que nous décomposons la contrainte interne `WITH COMPONENTS`. Il nous semble plus judicieux de décomposer seulement la contrainte interne sans nous préoccuper de la validité sémantique des contraintes nouvelles que nous formons — nous le ferons plus avant, en ne considérant que les contraintes simplifiées. Ainsi notre déclaration précédente se réécrit en :

```
T ::= SET {a INTEGER (0|1)(16)}
```

La même difficulté d'analyse se pose avec les contraintes par valeur (ITU, 1994a, § 45.2) de types structurés. Ce sera notre remarque suivante. Nous aurions ainsi pu avoir à analyser :

```
T ::= SET {a INTEGER (0|1)} ({a 16})
```

Nous allons réécrire d'abord cette déclaration en :

```
T ::= SET {
        a INTEGER (0|1)
      } ({a 16}) (WITH COMPONENTS {a (16)})
```

Il est clair que la contrainte interne produite ne change pas la sémantique. Ensuite nous appliquerons la réduction des contraintes internes que nous avons brièvement décrite précédemment :

```
T ::= SET {a INTEGER (0|1)(16)} ({a 16})
```

Nous expliquerons plus loin pourquoi nous conservons la contrainte `({a 16})`.

Il y a un troisième aspect qui méritera notre attention dans cette quatrième étape : les types récursifs. Les types du noyau d'ASN.1 sont bien fondés (voir la deuxième étape du chapitre 1 page 7), ce qui signifie que les types récursifs ont déjà été filtrés selon certains critères. Notre préoccupation ici est la terminaison de notre réécriture. Par exemple :

```
T ::= CHOICE {
        a [0] T (WITH COMPONENTS {..., b (0)}),
        b [1] INTEGER
      } (WITH COMPONENTS {..., b (0|1)})
```

Si l'on cherche, dans le but de réduire la contrainte `(WITH COMPONENTS {..., b (0)})`, à déplier la référence `T` à laquelle elle s'applique (champ `a`), alors nous faisons augmenter la taille du type et faisons apparaître une nouvelle contrainte interne :

```
T ::= CHOICE {
        a [0] CHOICE {
                a [0] T (WITH COMPONENTS {..., b (0)}),
                b [1] INTEGER
              } (WITH COMPONENTS {..., b (0|1)})
                (WITH COMPONENTS {..., b (0)}),
        b [1] INTEGER
      } (WITH COMPONENTS {..., b (0|1)})
```

Il est intuitivement clair que cette stratégie de réduction ne termine pas. Nous verrons donc comment procéder en présence de types récursifs.

Après cette courte introduction, nous allons donner le détail de cette quatrième étape, particulièrement complexe. Cette étape consiste en la répétition d'une suite de réécritures jusqu'à ce que la spécification soit invariante. (Pour les théoriciens des langages, cette étape est le point fixe d'une composition de réécritures.) L'idée est que lorsque l'invariance est atteinte, il n'y a plus de contraintes internes dans la spécification. Cette suite de réécritures est constituée de quatre étapes que nous détaillons maintenant.

1. *Réduction partielle des contraintes internes* (formellement à la section 9.5.1 page 220). Nous considérons chaque déclaration de sous-types, constituée à présent (après la seconde étape) d'une union de sous-types. Nous traitons séparément chaque terme (sous-type) de l'union, en leur appliquant deux opérations successivement :
   (a) *Produire les contraintes internes correspondant aux contraintes par valeurs* (formellement à la section 9.5.1 page 221). Nous avons abordé ce sujet en guise d'introduction de cette quatrième étape. Nous avions donné l'exemple :

   ```
   T ::= SET {a INTEGER (0|1)} ({a 16})
   ```

   que nous réécrivons ici en

   ```
   T ::= SET {
           a INTEGER (0|1)
         } ({a 16}) (WITH COMPONENTS {a (16)})
   ```

   Donnons les autres cas et les contraintes produites. Le cas du type `SET OF` (le cas `SEQUENCE OF` est identique) :

   ```
   T ::= SET ({}) OF INTEGER
   ```

   est réécrit en :

   ```
   T ::= SET ({}) (SIZE (0)) OF INTEGER
   ```

   Nous pouvons remarquer que nous produisons une contrainte de taille (`SIZE`), égale au nombre d'éléments que contient la contrainte de valeur (ici 0). Dans un cas plus général :

   ```
   T ::= SET ({4,7,2}) OF INTEGER
   ```

   est réécrit en :

   ```
   T ::= SET ({4,7,2}) (SIZE (3)) (WITH COMPONENT (4|7|2))
         OF INTEGER
   ```

   Ici, en plus de la contrainte de taille `(SIZE (3))`, nous avons produit une contrainte interne `(WITH COMPONENT (4|7|2))`. Nous comprenons pourquoi nous avons conservé la contrainte de valeur initiale `({4,7,2})`. En effet, les deux contraintes produites, en intersection entre elles et avec la première, sont strictement moins fortes que la contrainte initiale. Cela signifie que le type `T'` :

   ```
   T' ::= SET (SIZE(3))(WITH COMPONENT (4|7|2)) OF INTEGER
   ```

   contient strictement plus de valeurs que `T` : il suffit de constater, par exemple, que `{4,4,4}` est de type `T'` mais pas du type `T`. Donc il nous faut conserver la contrainte de valeur initiale pour que l'intersection de contraintes n'engendre pas un sous-type avec des valeurs en plus ou en moins (la sémantique doit rester invariante par nos réécritures). De façon très générale :

```
T ::= SET ({v1, ..., vn}) OF t
```

où les v_i dénotent des valeurs et t un type, devient :

```
T ::= SET ({v1, ..., vn})
          (SIZE (n))
          (WITH COMPONENT (v1 | ... | vn))
      OF t
```

Prenons un autre cas :

```
T ::= CHOICE {
        a INTEGER,
        b BOOLEAN
      } (a : 7)
```

est réécrit en :

```
T ::= CHOICE {
        a INTEGER,
        b BOOLEAN
      } (a : 7) (WITH COMPONENTS {a (7) PRESENT})
```

La contrainte interne produite ici est équivalente à la contrainte par valeur, mais nous conservons tout de même cette dernière par souci d'uniformité. Notons que la contrainte interne produite est complète (ITU, 1994a, § 45.8.6) et spécifie que le champ référencé `a` doit être présent. Voyons le cas du type `SET`. La spécification

```
T ::= SET {
        a INTEGER,
        b BOOLEAN OPTIONAL,
        c REAL OPTIONAL
      } ({a 7, b TRUE})
```

est réécrite en :

```
T ::= SET {
        a INTEGER,
        b BOOLEAN OPTIONAL,
        c REAL OPTIONAL
      } ({a 7, b TRUE})
        (WITH COMPONENTS {a (7),
                          b (TRUE) PRESENT,
                          c ABSENT})
```

La contrainte interne étant complète, tous les champs doivent y apparaître, et, s'ils sont optionnels, nous indiquons leur présence ou absence. Dans le sous-cas :

```
T ::= SET {} ({})
```

nous laissons invariante la spécification.

Le cas du type `SEQUENCE` est identique au type `SET`, à une exception près : le type `REAL`. En effet, lors de la première étape de la mise sous forme canonique (au chapitre 1 page 7), nous avions décidé de définir le type `REAL` de la façon suivante :

```
REAL ::= SEQUENCE {
           mantissa INTEGER,
           base     INTEGER (2|10),
           exponent INTEGER
         }
```

Ainsi, nous pouvons traiter les contraintes du type `REAL` comme des contraintes internes normales :

```
T ::= REAL (WITH COMPONENTS {..., base (2)})
```

Mais nous ne produirons pas de contraintes internes à partir des contraintes par valeurs pour le type `REAL`. En effet :

```
T ::= REAL ({mantissa 1, base 2, exponent 2})
           ({mantissa 2, base 2, exponent 1})
```

est une spécification correcte, mais

```
T ::= SEQUENCE {
        mantissa INTEGER (1) (2),
        base     INTEGER (2|10) (2) (2),
        exponent INTEGER (2) (1)
      } ({mantissa 1, base 2, exponent 2})
        ({mantissa 2, base 2, exponent 1})
```

ne l'est pas. Les valeurs dans les contraintes du type `REAL` seront donc traitées à part.

(b) *Réduire les contraintes internes* (formellement à la section 9.5.1 page 222). Après avoir produit des contraintes internes supplémentaires à partir des contraintes par valeurs, nous réduisons *partiellement* les contraintes internes de la spécification. Cela ne signifie pas qu'à la fin de cette réécriture nous n'avons plus du tout de contraintes internes. En effet, il est possible qu'une contrainte interne contienne une autre contrainte interne, donc ce que nous faisons ici est réduire toutes les contraintes internes (elles apparaissent au niveau global, comme toutes les contraintes du noyau d'ASN.1) *une seule fois*, c'est-à-dire, en termes de programmation : sans appels récursifs. Voyons cas par cas le procédé. Tout d'abord le cas du type `SET OF` (le cas du type `SEQUENCE OF` est identique) :

```
T ::= SET (WITH COMPONENT (0|1)) OF INTEGER
```

est réécrit en :

```
T ::= SET OF (INTEGER (0|1))
```

Il est très important de noter que nous avons fait apparaître une contrainte (ici, `(0|1)`) *dans* un type structuré, ce qui fait que le sous-type résultant de cette réécriture n'est pas dans le noyau d'ASN.1, où toutes les contraintes sont au niveau global. Nous verrons plus tard comment rendre ces sous-types conforme au noyau.

Toujours à l'aide du type `SET OF`, donnons maintenant un exemple exotique de contrainte interne contenant une autre contrainte interne :

```
T  ::= SET (WITH COMPONENT (WITH COMPONENT (0|1))) OF #1
#1 ::= SET OF #2
#2 ::= INTEGER
```

Cette spécification est tout à fait conforme au noyau d'ASN.1. Ici nous réécrivons en :

```
T  ::= SET OF (#1 (WITH COMPONENT (0|1)))
#1 ::= SET OF #2
#2 ::= INTEGER
```

Nous avons utilisé les parenthèses supplémentaires que nous autorise le noyau d'ASN.1 pour mieux séparer ou regrouper les types et les contraintes. Nous constatons que nous avons toujours une contrainte interne. C'est, avec la possibilité des types récursifs, ce qui nous fait répéter la séquence de réécritures dont nous décrivons ici une sous-étape.

Pour le cas du type `CHOICE` il y a plusieurs sous-cas.

i. *Un champ du* ***`CHOICE`*** *est référencé dans la contrainte, où sa présence n'est pas contrainte.* Par exemple :

```
T ::= CHOICE {
        a INTEGER,
        b BOOLEAN
      } (WITH COMPONENTS {..., a (0|1)})
```

Alors nous réécrivons :

```
T ::= CHOICE {
        a INTEGER (0|1),
        b BOOLEAN
      }
```

ii. *Un champ du* ***`CHOICE`*** *est référencé dans la contrainte, où sa présence est spécifiée* ***`PRESENT`***. Par exemple :

```
T ::= CHOICE {
        a INTEGER,
        b BOOLEAN
      } (WITH COMPONENTS {..., a (0|1) PRESENT})
```

devient :

```
T ::= CHOICE {a INTEGER (0|1)}
```

Au passage nous nous sommes assuré qu'aucun autre champ n'était spécifié `PRESENT` (ITU, 1994a, § 45.8.9.2).

iii. *Un champ du* ***`CHOICE`*** *est référencé dans la contrainte, où sa présence est spécifiée* ***`ABSENT`***. Par exemple :

```
T ::= CHOICE {
        a INTEGER,
        b BOOLEAN
      } (WITH COMPONENTS {..., a (0|1) ABSENT})
```

est réécrit en

```
T ::= CHOICE {b BOOLEAN}
```

Remarquons deux choses :

— Nous nous sommes débarrassé de la contrainte `(0|1)`. La syntaxe d'ASN.1 nous permet de la spécifier, mais elle n'est pas significative (ce qui n'apparaît pas explicitement dans X.680 (ITU, 1994a)).

— Nous nous sommes assurés au passage qu'il y avait au moins un autre champ dans le `CHOICE` : ici `b`. Nous comblons ainsi une lacune de la norme X.680 (ITU, 1994a).

iv. *Un champ du `CHOICE` n'est pas référencé dans la contrainte.* Dans ce cas on examine les autres champs du type.

v. *Si la contrainte interne est complète, alors les champs non référencés dans la contrainte sont ôtés du type.* Par exemple :

```
T ::= CHOICE {
        a INTEGER,
        b BOOLEAN
      } (WITH COMPONENTS {a (0|1)})
```

est réécrit en :

```
T ::= CHOICE {a INTEGER (0|1)}
```

Nous satisfaisons ainsi (ITU, 1994a, § 45.8.6).

Examinons maintenant le cas du type `SEQUENCE`.

i. *Un champ `OPTIONAL` est contraint `ABSENT`.* Par exemple :

```
T ::= SEQUENCE {
        a INTEGER OPTIONAL,
        b BOOLEAN
      } (WITH COMPONENTS {..., a (0|1) ABSENT})
```

est réécrit en :

```
T ::= SEQUENCE {b BOOLEAN}
```

Nous nous sommes débarrassé de la contrainte `(0|1)`. La syntaxe d'ASN.1 nous permet de la spécifier, mais elle n'est pas significative (ce qui n'apparaît pas explicitement dans X.680 (ITU, 1994a)).

ii. *Une contrainte partielle ne contraint pas la présence d'un champ qu'elle référence.* Dans ce cas, on réduit la contrainte interne sans se poser de questions. Par exemple :

```
T ::= SEQUENCE {
        a INTEGER,
        b BOOLEAN
      } (WITH COMPONENTS {..., a (0|1)})
```

est réécrite en :

```
T ::= SEQUENCE {
        a INTEGER (0|1),
        b BOOLEAN
      }
```

Rappelons que les champs des types du noyau d'ASN.1 ne possèdent pas de contraintes.

iii. *Une contrainte complète ne contraint pas la présence d'un champ* `OPTIONAL` *qu'elle référence.* Dans ce cas nous réécrivons la contrainte en imposant `PRESENT` (conformément à (ITU, 1994a, § 45.8.9.3.a)). Par exemple :

```
T ::= SEQUENCE {
        a INTEGER OPTIONAL
      } (WITH COMPONENTS  {a (0|1)})
```

devient :

```
T ::= SEQUENCE {
        a INTEGER OPTIONAL
      } (WITH COMPONENTS  {a (0|1) PRESENT})
```

iv. *Une contrainte* `OPTIONAL` *contraint un champ* `OPTIONAL`*.* Dans ce cas nous réduisons la contrainte interne sans plus de questions. Par exemple :

```
T ::= SEQUENCE {
        a INTEGER OPTIONAL
      } (WITH COMPONENTS {a (0|1) OPTIONAL})
```

devient :

```
T ::= SEQUENCE {a INTEGER (0|1) OPTIONAL}
```

v. *Une contrainte* `PRESENT` *s'applique à un champ* `OPTIONAL`*.* Dans cas nous réduisons la contrainte interne et nous ôtons l'attribut `OPTIONAL` du champ sur lequel elle s'applique. Par exemple :

```
T ::= SEQUENCE {
        a INTEGER OPTIONAL,
        b BOOLEAN
      } (WITH COMPONENTS {a (0|1) PRESENT})
```

devient :

```
T ::= SEQUENCE {
        a INTEGER (0|1),
        b BOOLEAN
      }
```

vi. *La contrainte interne est partielle et des champs n'y sont pas contraints.* Dans ce cas ces champs sont conservés tels quels, conformément à (ITU, 1994a, § 45.8.6).

vii. *La contrainte interne est complète et un champ* `OPTIONAL` *n'est pas référencé.* Dans ce cas, nous ajoutons une contrainte `ABSENT` sur ce champ, conformément à (ITU, 1994a, § 45.8.6). Par exemple :

```
T ::= SEQUENCE {
        a INTEGER OPTIONAL,
        b BOOLEAN
      } (WITH COMPONENTS {b (TRUE)})
```

devient :

```
T ::= SEQUENCE {
        a INTEGER OPTIONAL,
        b BOOLEAN
      } (WITH COMPONENTS {a ABSENT, b (TRUE)})
```

Remarquons qu'il faut ensuite chercher à appliquer un autre cas pour traiter la contrainte que nous venons de rajouter (en termes de programmation, cela revient à un appel récursif).

viii. *La contrainte interne est complète et un champ non* `OPTIONAL` *n'y est pas référencé.* Dans ce cas le champ est inchangé, conformément à (ITU, 1994a, § 45.8.6).

Rappelons que les cas manquants ci-dessus, et, de façon générale, dans toute énumération de cas, sont des cas d'erreur. Par ailleurs, remarquons ici que la norme ne permet pas les contraintes vides, alors que leur définition peut s'appliquer uniformément au cas vide. Par exemple :

```
T ::= SEQUENCE {
        a INTEGER OPTIONAL,
        b BOOLEAN
      } (WITH COMPONENTS {})
```

est syntaxiquement interdit, alors qu'on pourrait légitimement et sans effort aucun lui donner le sens :

```
T ::= SEQUENCE {
        a INTEGER OPTIONAL,
        b BOOLEAN
      } (WITH COMPONENTS {a ABSENT})
```

d'après (ITU, 1994a, § 45.8.6). Dans le cas d'une contrainte partielle, la spécification serait alors invariante (ce qui n'est pas un problème du tout). Par exemple :

```
T ::= SEQUENCE {
        a INTEGER OPTIONAL,
        b BOOLEAN
      } (WITH COMPONENTS {...,})
```

pourrait être interprété comme :

```
T ::= SEQUENCE {
        a INTEGER OPTIONAL,
        b BOOLEAN
      }
```

en accord avec le même paragraphe (ITU, 1994a, § 45.8.6). Insistons que ces cas, pour exotiques qu'ils puissent paraître, n'en sont pas moins pertinents du point de vue ensembliste (sous-jacent au sous-typage d'ASN.1), pour lequel l'ensemble vide est un ensemble à part entière.

Le cas du type `SET` est identique à celui du type `SEQUENCE`, excepté le fait — sur lequel nous n'avons pas insisté — que l'ordre des contraintes n'est pas significatif. À ce sujet, rappelons que (ITU, 1994a, § 45.8.5) impose que les contraintes dans la contrainte interne apparaissent dans le même ordre que les champs sur lesquels elles s'appliquent. Dans cette thèse nous avons respecté cette restriction, mais nous soutenons, comme pour l'ordre des valeurs constituant une valeur de type `SEQUENCE` (ITU, 1994a, § 22.14), qu'elle est fondamentalement inutile (car contrairement à X.208 (ITU, 1990), les contraintes dans une contrainte interne doivent référencer explicitement le champ sur lequel elles s'appliquent, et ces champs ont toujours un label) et, si elle n'alourdit pas vraiment la norme, elle complique significativement sa formalisation.

2. *Globalisation des types locaux sans leur étiquetage* (formellement à la section 6.4.1 page 122). Après avoir réduit partiellement les contraintes internes les plus externes, nous avons peut-être engendré des sous-types qui n'appartiennent pas au noyau d'ASN.1. Par exemple :

```
#0 ::= INTEGER
T  ::= SET {
         a #0
       } (WITH COMPONENTS {a (0|1)})
```

a été réécrit en :

```
#0 ::= INTEGER
T  ::= SET {a #0 (0|1)}
```

Or ce dernier sous-type est un type stucturé dont un des champs au moins possède des contraintes de sous-typage : il n'est donc pas dans le noyau d'ASN.1 tel que nous l'avons présenté au chapitre 1

page 7, et défini au chapitre 6 page 119. Nous allons simplement lui appliquer à nouveau la réécriture de la cinquième étape du chapitre 1 page 7. Ainsi, nous poursuivons :

```
#0 ::= INTEGER
T  ::= SET {a #1}
#1 ::= #0 (0|1)
```

Quel est l'intérêt de poursuivre ainsi ? En plus de chercher à revenir dans le noyau d'ASN.1, l'intérêt est que la contrainte interne initiale contenait peut-être une contrainte interne, qu'il faudra donc traiter lors d'une prochaine itération. Reprenons un exemple donné plus haut :

```
T  ::= SET (WITH COMPONENT (WITH COMPONENT (0|1))) OF #1
#1 ::= SET OF #2
#2 ::= INTEGER
```

Nous avons réécrit cette spécification en

```
T  ::= SET OF (#1 (WITH COMPONENT (0|1)))
#1 ::= SET OF #2
#2 ::= INTEGER
```

Maintenant, lors de la présente sous-étape, nous réécrivons :

```
T  ::= SET OF #3
#1 ::= SET OF #2
#2 ::= INTEGER
#3 ::= #1 (WITH COMPONENT (0|1))
```

De cette façon nous avons placé la contrainte interne qui restait à la position qui convient pour reprendre un peu plus tard le processus que nous présentons (récursivement).

3. *Dépliage des abréviations globales* (formellement à la section 6.4.3 page 125). Dans le but de ramener les sous-types produits par la réduction partielle des contraintes internes dans le noyau d'ASN.1, nous poursuivons en dépliant les abréviations globales que nous avons peut-être engendrées à la sous-étape précédente. En reprenant notre dernier exemple, il vient alors :

   ```
   T  ::= SET OF #3
   #1 ::= SET OF #2
   #2 ::= INTEGER
   #3 ::= (SET OF #2) (WITH COMPONENT (0|1))
   ```

   Nous avons remplacé l'abréviation `#1` dans la définition de `#3` par sa définition (`SET OF #2`). Notez les parenthèses supplémentaires, typiques du noyau d'ASN.1.

4. *Distribution des contraintes (disjonctives)* (formellement à la section 9.3 page 219). Il est possible que les nouvelles contraintes,

bien qu'elles soient obligatoirement en forme disjonctive, ne soient pas distribuées sur les types sur lesquels elles s'appliquent : nous les distribuons donc, exactement comme nous l'avions fait à la seconde étape de ce présent chapitre. Par exemple :

```
#1 ::= VisibleString
T  ::= SET {a #1}
           (WITH COMPONENTS {a (SIZE (5) | FROM ("A".."B"))})
```

La première sous-étape réduit la contrainte interne :

```
#1 ::= VisibleString
T  ::= SET {a #1 (SIZE (5) | FROM ("A".."B"))}
```

La seconde sous-étape sort les contraintes nouvellement apparues sur les champs :

```
#1 ::= VisibleString
T  ::= SET {a #2}
#2 ::= #1 (SIZE (5) | FROM ("A".."B"))
```

La troisième sous-étape déplie l'abréviation `#1` :

```
#1 ::= VisibleString
T  ::= SET {a #2}
#2 ::= VisibleString (SIZE (5) | FROM ("A".."B"))
```

La présente quatrième sous-étape distribue la contrainte (forcément disjonctive car la contrainte interne (initiale) était en forme disjonctive) :

```
#1 ::= VisibleString
T  ::= SET {a #2}
#2 ::= VisibleString (SIZE (5))
     ∪ VisibleString (FROM ("A".."B"))
```

Nous nous arrêtons car la spécification est alors invariante par la présente quatrième étape. (Nous avons atteint le point fixe.)

Donnons deux exemples supplémentaires où les sous-étapes précédentes sont itérées. Commençons par :

```
T ::= CHOICE {
        a [0] T (WITH COMPONENTS {..., b (0)}),
        b [1] INTEGER
      } (WITH COMPONENTS {..., b (0|1)})
```

Au chapitre 1 page 7 cette déclaration est successivement réécrite en :

```
T  ::= CHOICE {
         a #1,
         b #2
       } (WITH COMPONENTS {..., b (0|1)})
#1 ::= [0] T (WITH COMPONENTS {..., b (0)})
#2 ::= [1] INTEGER
```

puis :

```
T  ::= CHOICE {
         a #1,
         b #2
       } (WITH COMPONENTS {..., b (0|1)})
#1 ::= [0] CHOICE {
         a #1,
         b #2
       } (WITH COMPONENTS {..., b (0|1)})
         (WITH COMPONENTS {..., b (0)})
#2 ::= [1] INTEGER
```

Maintenant, la première sous-étape consiste à réduire partiellement les contraintes internes (les plus externes). Nous réécrivons donc :

```
T  ::= CHOICE {
         a #1,
         b #2 (0|1)
       }
#1 ::= [0] CHOICE {
         a #1,
         b #2 (0|1) (0)
       }
#2 ::= [1] INTEGER
```

Il est utile de remarquer que l'ordre dans lequel nous réduisons les contraintes internes en intersection *n'est pas* significatif, car l'opérateur d'intersection (^) commute.

Ensuite, la deuxième sous-étape sort les contraintes des champs :

```
T  ::= CHOICE {
         a #1,
         b #3
       }
#1 ::= [0] CHOICE {
         a #1,
         b #4
       }
#2 ::= [1] INTEGER
#3 ::= #2 (0|1)
#4 ::= #2 (0|1) (0)
```

Ensuite, la troisième sous-étape déplie les abréviations :

```
T  ::= CHOICE {
         a #1,
         b #3
       }
```

```
#1 ::= [0] CHOICE {
         a #1,
         b #4
       }
#2 ::= [1] INTEGER
#3 ::= [1] INTEGER (0|1)
#4 ::= [1] INTEGER (0|1)(0)
```

Envisageons maintenant un second exemple. Si en plus de la déclaration de `T` précédente, nous avions eu en plus :

```
U ::= T (WITH COMPONENTS {..., a (WITH COMPONENTS {..., b (0)})})
```

alors tout d'abord le type `U` aurait été mis en forme canonique, en dépliant l'abréviation `T` :

```
U ::= CHOICE {
        a #1,
        b #3
      } (WITH COMPONENTS {..., a (WITH COMPONENTS {..., b (0)})})
```

Puis la première sous-étape aurait réduit partiellement les contraintes internes :

```
U ::= CHOICE {
        a #1 (WITH COMPONENTS {..., b (0)}),
        b #3
      }
```

Ensuite la seconde sous-étape aurait sorti les sous-types des champs :

```
U  ::= CHOICE {
         a #5,
         b #3
       }
#5 ::= #1 (WITH COMPONENTS {..., b (0)})
```

Puis la troisième sous-étape aurait déplié `#1` :

```
U  ::= CHOICE {
         a #5,
         b #3
       }
#5 ::= [0] CHOICE {
         a #1,
         b #4
       } (WITH COMPONENTS {..., b (0)})
```

La quatrième et dernière sous-étape n'aurait rien fait car les contraintes sont bien distribuées. Mais il reste encore une contrainte interne à réduire et nous aurions alors recommencé la présente quatrième étape. D'abord en réduisant la contrainte interne :

```
U  ::= CHOICE {
         a #5,
         b #3
       }
#5 ::= [0] CHOICE {
         a #1,
         b #4 (0)
       }
```

Ensuite en sortant les champs contraints :

```
U  ::= CHOICE {
         a #5,
         b #3
       }
#5 ::= [0] CHOICE {
         a #1,
         b #6
       }
#6 ::= #4 (0)
```

Puis en dépliant les abréviations globales nouvellement apparues (ici `#4`) :

```
U  ::= CHOICE {
         a #5,
         b #3
       }
#5 ::= [0] CHOICE {
         a #1,
         b #6
       }
#6 ::= [1] INTEGER (0|1)(0)(0)
```

Et nous aurions alors terminé car la spécification est maintenant invariante pour la quatrième et présente étape.

Avant de refermer cette longue et quatrième étape, disons un mot sur la terminaison de ces réécritures. La différence essentielle entre les types (et sous-types) et les contraintes est que les premiers peuvent être nommés et pas les seconds. Or, comme le système de types d'ASN.1 n'inclut pas d'opérateurs de point fixe, les types et les sous-types peuvent être récursifs mais pas les contraintes. Pour prouver la terminaison de notre algorithme nous utiliserions cette propriété.

La **cinquième étape** consiste à vérifier que les sous-types engendrés par les précédentes étapes sont bien fondés. Lors de la deuxième étape de la mise en forme des modules ASN.1 (au chapitre 1 page 7), nous avions déjà effectué cette vérification, qui consiste essentiellement à rejeter certaines catégories de types récursifs. Cette vérification ignore les contraintes de sous-typage, aussi nous avions dit que

```
T ::= CHOICE {
        a [0] T,
        b NULL
       } (WITH COMPONENTS {b ABSENT})
```

était un type bien fondé, alors que

```
T ::= CHOICE {a [0] T}
```

ne l'était pas. Ici, après avoir réduit toutes les contraintes internes en contraintes élémentaires, nous allons appliquer à nouveau le critère des types bien fondés, et ainsi rejeter cette précédente déclaration qui jusqu'à présent était valide.

La **sixième étape** (voir la contrepartie formelle à la section 9.6 page 224) traite des intersections de contraintes de tailles (`SIZE`). Pourquoi traiter séparément ces contraintes ? La raison réside dans l'interaction entre les contraintes `SIZE` et `FROM`. Par exemple :

```
T ::= VisibleString (FROM ("A".."C")) (FROM ("D".."F"))
```

est un sous-type qui ne contient pas de valeurs, donc qui doit être rejeté. Mais

```
T ::= VisibleString (SIZE(0)) (FROM ("A".."C")) (FROM ("D".."F"))
```

est un sous-type valide car il contient une unique valeur qui est la chaîne vide (`""`). Il s'agit donc de calculer l'intersection des contraintes `SIZE` avant les autres, et ensuite, si la taille calculée se ramène à zéro (ce peut être `0..0` ou `-1<..0` etc.) d'éliminer les éventuelles contraintes `FROM` de l'intersection, avant de réduire toute l'intersection.

La **première sous-étape** (voir la contrepartie formelle à la section 9.7.2 page 227) consiste à réduire les contraintes `SIZE` en appliquant la règle de réécriture suivante à toute paire de contraintes `(SIZE (`σ_0`))` et `(SIZE (`σ_1`))` en intersection :

$$\texttt{(SIZE (}\sigma_0\texttt{))}\ \texttt{\^{}}\ \texttt{(SIZE (}\sigma_1\texttt{))} \longrightarrow \texttt{(SIZE (}\sigma_0\ \texttt{\^{}}\ \sigma_1\texttt{))},$$

où σ_0 et σ_1 dénotent des contraintes et `^` est l'intersection de contraintes. Par exemple :

```
T ::= VisibleString (SIZE (0..3)) (SIZE (2))
```

est réécrit en :

```
T ::= VisibleString (SIZE ((0..3)(2)))
```

L'ordre des intersections nouvellement formées n'est pas significatif. Ainsi il aurait été correct de réécrire :

```
T ::= VisibleString (SIZE ((2)(0..3)))
```

De même, il est correct de définir :

```
T ::= VisibleString (SIZE (MIN..5)) (SIZE (2))
```

et de réécrire en :

```
T ::= VisibleString (SIZE ((MIN..5)(2)))
```

ou en :

```
T ::= VisibleString (SIZE ((2)(MIN..5)))
```

La **seconde sous-étape** (voir la contrepartie formelle à la section 9.6.2 page 225) va donc consister à rendre commutatives les intersections présentes dans les contraintes `SIZE`. Pour cela, nous allons remplacer, *lorsque cela est possible*, les éventuelles bornes `MIN` et `MAX` par leur valeur, en accord avec la clause (ITU, 1994a, § 45.4.4). (Les bornes `MINUS-INFINITY` ou `PLUS-INFINITY` ne sont pas possibles ici car les intervalles doivent être du type `INTEGER`. Nous parlerons plus loin de ces bornes réelles spéciales lorsque nous aborderons le cas général de la réduction des intersections d'intervalles.) Il n'est pas possible de remplacer les bornes `MIN` et `MAX` lorsqu'elles désignent respectivement l'infini négatif ($-\infty$) et l'infini positif ($+\infty$). Maintenant il y a au plus une contrainte `SIZE` par sous-type.

La **troisième sous-étape** normalise les intervalles éventuellement présents dans les contraintes `SIZE` : nous les transformons en intervalles fermés, et s'ils sont réduits à un élément, alors nous les remplaçons par une contrainte par valeur. Par exemple :

```
U ::= VisibleString (SIZE (-1<..3)) (SIZE (0<..1))
```

avait été réécrit à la première sous-étape en :

```
U ::= VisibleString (SIZE ((-1<..3) (0<..1)))
```

Maintenant nous réécrivons :

```
U ::= VisibleString (SIZE ((0..3) (1)))
```

La **quatrième sous-étape** (voir la contrepartie formelle à la section 9.6.3 page 225) consiste à vérifier que les intervalles fermés sont bien formés, c'est-à-dire que ce sont bien des intervalles au sens mathématique. Par exemple :

```
A ::= INTEGER (5..4)
B ::= INTEGER (MAX..MIN)
```

sont mal formés.

La **cinquième sous-étape** (voir la contrepartie formelle à la section 9.7.1 page 227) calcule l'intersection des intervalles présents dans les `SIZE`. Ainsi

```
T ::= VisibleString (SIZE ((MIN..5)(2)))
```

devient :

```
T ::= VisibleString (SIZE (2))
```

Si nous avions eu :

```
T ::= VisibleString (SIZE ((0..3)(4..7)))
```

alors nous n'aurions pas pu aboutir, ce qui équivaut à une erreur, donc à rejeter le sous-type car il est vide (nous dirons volontiers qu'il est mal formé). Notre exemple

```
U ::= VisibleString (SIZE ((0..3)(1)))
```

devient quant à lui :

```
U ::= VisibleString (SIZE (1))
```

Il est très important de noter que pour pouvoir effectuer cette étape il faut s'assurer que la clause (ITU, 1994a, § 45.5.3) est satisfaite, ce qui fait appel au cas général que nous verrons plus avant (réduction générale des sous-types pour s'assurer de leur bonne formation). Étant donné `(SIZE (`σ`))`, où σ est une contrainte, nous devons contrôler le sous-type `INTEGER (0..MAX)(`σ`)`.

La **sixième sous-étape** (voir la contrepartie formelle à la section 9.7.3 page 228) examine les valeurs des contraintes `SIZE` et si elles valent zéro alors les éventuelles contraintes `FROM` en intersection sont éliminées. Par exemple :

```
T ::= VisibleString (SIZE(0)) (FROM ("A".."C")) (FROM ("D".."F"))
```

devient :

```
T ::= VisibleString (SIZE (0))
```

La **septième étape** consiste à traiter tous les intervalles et les contraintes par valeurs comme nous l'avons fait précédemment dans le cas des intervalles apparaissant dans les contraintes de taille `SIZE`. Ajoutons ici le cas où les valeurs et les intervalles sont de type `REAL`. Deux sous-cas sont possibles.

— *La valeur, éventuellement en tant que borne d'intervalle, n'est ni* ***`PLUS-INFINITY`*** *ni* ***`MINUS-INFINITY`***. Alors nous devons la normaliser pour permettre une comparaison aisée de ces valeurs. Par exemple les nombres `{mantissa 1, base 2, exponent 2}` et `{mantissa 2, base 2, exponent 1}` représentent le même nombre réel mathématique. Une fonction de normalisation sur les réels ASN.1 est une fonction qui associe le même réel ASN.1 à tous les réels ASN.1

qui dénotent le même réel mathématique. Les nombres réels normalisés doivent être exprimés en base 10, sinon il risque d'y avoir une perte de précision en partant d'une représentation en base 2. Voir la contrepartie formelle à la section 9.6.1 page 224.

— *La borne d'intervalle est `MIN` ou `MAX`.* Nous appliquons alors la même procédure que nous avons vu précédemment pour les intervalles d'entiers dans les `SIZE`. Si, après réécriture, la borne reste invariante, alors si elle est `MIN` on lui substitue `MINUS-INFINITY`, si elle est `MAX` alors on lui substitue `PLUS-INFINITY`. Voir la contrepartie formelle à la section 9.6.2 page 225.

Avant de passer à l'étape suivante, ajoutons un mot concernant les intersections de contraintes par valeurs. La règle que nous appliquerons pour calculer ces intersections est purement syntaxique et est simplement l'identité. Par exemple :

```
T ::= INTEGER (7)(7)
```

Dans ce cas il est clair que `7` et `7` sont deux portions de texte identiques, donc l'intersection est `7`. Mais que dire de :

```
T ::= (SET OF INTEGER) ({7,5}) ({5,9})
```

Dans ce cas, l'intersection est vide, et donc le sous-type est mal formé.

Une limitation de cette règle est qu'elle ne permet pas de conclure que

```
T ::= (SET OF INTEGER) ({7,5}) ({5,7})
```

est correct, car elle ne tient pas compte d'une permutation possible des valeurs de l'ensemble. Une solution simple consiste, lors de mise sous forme canonique des spécifications (au chapitre 6 page 119), d'appliquer un ordre canonique sur les valeurs éléments des `SET OF`, ainsi que sur les champs des `SET` (l'ordre alphabétique croissant sur les labels est suffisant). La définition d'un ordre canonique sur les valeurs ASN.1 ne posant pas de problèmes particuliers, nous ne jugeons pas nécessaire de la donner ici.

La **huitième étape** traite *isolément* chaque contrainte `FROM`. Comme pour les contraintes `SIZE` pour lesquelles nous avions vérifié la propriété (ITU, 1994a, § 45.5.3), nous devons vérifier la propriété (ITU, 1994a, § 45.7.3) pour les `FROM`. Ainsi, étant donné dans le cas général `T (FROM (`σ`))`, où σ est une contrainte, cela signifie que nous devons contrôler le sous-type `T(`σ`)`.

La **neuvième étape** (voir la contrepartie formelle à la section 9.7.5 page 229) est la dernière étape du contrôle des sous-types. Elle consiste à calculer l'intersection des contraintes que nous avons réécrites et filtrées jusqu'à présent. Les intersections entre contraintes de nature différentes

ne sont pas à calculer, sauf les contraintes par valeur qui sont de même nature que celles par intervalles — Une valeur est un cas dégénéré d'intervalle réduit à un élément. Les intersections de contraintes `SIZE` ont été calculées précédemment. Il nous reste alors à calculer les intersections d'intervalles et de valeurs, ce que nous avons déjà présenté lors des calculs sur les contraintes `SIZE`. Le seul cas restant est l'intersection de contraintes `FROM` : c'est exactement le même traitement que pour les intersections de contraintes `SIZE`. (La raison théorique est que si $\mathcal{A}$ et $\mathcal{B}$ sont des alphabets finis, alors nous avons : $\mathcal{A}^* \cap \mathcal{B}^* = (\mathcal{A} \cap \mathcal{B})^*$, où $\mathcal{X}^*$ est le sous-monoïde engendré par $\mathcal{X}$.) Par exemple,

```
T ::= VisibleString (FROM ("A".."C")) (FROM ("B".."D"))
```

est réécrit en :

```
T ::= VisibleString (FROM (("A".."C") ("B".."D")))
```

puis

```
T ::= VisibleString (FROM ("B".."C"))
```

De façon générale les seules réécritures sur les contraintes `SIZE` et `FROM` qui laissent invariante la sémantique (intuitivement cela signifie que le sous-type contient alors les mêmes valeurs avant et après la réécriture) sont les suivantes (on tiendra compte en plus de la commutativité des opérateurs `|` et `^`) :

`(FROM (`σ_0`)) ^ (FROM (`σ_1`))` $\longleftrightarrow$ `(FROM (`σ_0 `^` σ_1`))`
`(SIZE (`σ_0`)) ^ (SIZE (`σ_1`))` $\longleftrightarrow$ `(SIZE (`σ_0 `^` σ_1`))`
`(SIZE (`σ_0`) | SIZE (`σ_1`))` $\longleftrightarrow$ `(SIZE (`σ_0 `|` σ_1`))`
`(FROM (`σ_0`) | FROM (`σ_1`))` $\longleftrightarrow$ `(FROM (`σ_1`))`, si $\sigma_0 \subseteq \sigma_1$.

Cette dernière réécriture s'illustre par l'exemple :

```
T ::= VisibleString (FROM ("A") | FROM ("A".."B"))
```

qui se réécrit :

```
T ::= VisibleString (FROM ("A".."B"))
```

(En théorie des langages formels on démontrerait le théorème : $\mathcal{A}^* + \mathcal{B}^* \subseteq (\mathcal{A} + \mathcal{B})^*$.)

Les règles de codage PER (ITU, 1994f) sauraient tirer profit de cette dernière réécriture, mais dans cette thèse nous ne sommes pas allé jusque là, car notre souci principal était la vérification sémantique et non la compilation, même si pour parvenir à la vérification nous avons choisi de réécrire les spécifications, et donc à les précompiler.

Conclusion

En entreprenant cette thèse notre objectif était double. D'une part il s'agissait d'appliquer une certaine logique mathématique, normalement employée en théorie des langages de programmation, comme méthode formelle de spécification pour ASN.1. D'autre part il était important que cette recherche aboutisse à la réalisation d'un outil logiciel fondé sur nos résultats théoriques et qui s'adresse aux télécommunicants dans l'industrie.

L'intérêt de l'emploi en tant que formalisme logique de la sémantique opérationnelle structurée est que celle-ci se prête bien aux preuves inductives (analyse par cas) et permet d'exprimer tout aussi bien des algorithmes — éliminant ainsi une barrière entre spécification et algorithmique.

Nous avons couvert tous les aspects d'ASN.1, tel que ce langage est normalisé dans X.680 (ITU, 1994a), sans concession d'aucune sorte. Nous avons établi une définition formelle d'ASN.1 en suivant une démarche pragmatique et expérimentale, validée par une fertile maïeutique avec Olivier Dubuisson (France Télécom (CNET), Lannion), Bancroft Scott (OSS Inc. & ISO), Paul Thorpe (OSS Inc.) et Heinz Schaefer (Siemens, Vienne, Autriche). Ce faisant nous avons ainsi mis au jour de grandes et coûteuses maladresses de conception d'ASN.1, ainsi que des incomplétudes importantes de la norme. Nous avons contribué auprès de l'ISO, via l'AFNOR, à l'amélioration de celle-ci.

En formalisant un codage de référence, abstrait à partir de X.690 (BER) (ITU, 1994e), nous avons pu munir la spécification syntaxique d'une contrepartie sémantique. Nous avons démontré que, pour un noyau d'ASN.1 que nous avons défini rigoureusement, la composition du codage et du décodage est l'identité. Ce résultat implique la correction sémantique du codage de référence et est donc une contribution importante quant à la sûreté du protocole ASN.1/BER. Nous avons énoncé et prouvé deux théorèmes qui établissent la non ambigüité des contrôles des types (syntaxiques et sémantiques), c'est-à-dire, en termes plus opérationnels, le déterminisme des algorithmes qui leur sont associés. Ces résultats théo-

riques fournissent *a posteriori* une justification et une interprétation de certaines contraintes sur les types ASN.1 : les contrôles (syntaxiques et sémantiques) des types, s'ils réussissent, ne peuvent le faire que d'une seule façon. En d'autres termes encore, cela implique qu'un éventuel décodage fondé sur ces contrôles s'exécutera sans rebroussements, donc efficacement. Cela implique aussi que l'analyse par cas de notre spécification formelle peut être reprise telle quelle pour écrire les algorithmes de contrôle (syntaxique et sémantique) des types.

Notre recherche a été réalisée à l'INRIA-Rocquencourt (Institut National de la Recherche en Informatique et Automatique), dans le projet Cristal, dont les thèmes d'investigation sont la programmation typée et les compilateurs. L'outil logiciel par excellence y est le système Caml qui y est développé, offrant un langage de programmation et des compilateurs très expressifs et sûrs. En particulier OCaml est un langage fonctionnel impur de la famille ML, fortement et statiquement typé, modulaire, à objets et valeurs polymorphes, conjuguant l'inférence de type à la compilation, la compilation séparée ainsi que d'excellentes performances (en particulier en code natif). C'est ainsi que l'analyseur de protocoles ASN.1, fruit de notre thèse, est réalisé en OCaml. Son nom est Asno et c'est un contrôleur de spécifications ASN.1 dont l'ergonomie a été particulièrement soignée (en particulier la signalisation des erreurs) pour permettre une utilisation aisée de la part des industriels. Encore un avantage de l'emploi de la sémantique opérationnelle est ici que la sémantique du langage OCaml est elle-même spécifiée dans ce formalisme et qu'inversement OCaml se prête tout particulièrement à l'implémentation de ces spécifications formelles.

Même la syntaxe d'ASN.1 (dans sa version X.208 (ITU, 1990)) n'a pas été laissée de côté. Celle-ci est normalisée mais est hautement inadéquate pour une génération automatique d'analyseurs syntaxiques (à l'aide d'outils comme Yacc ou Bison). Nous avons alors fourni une grammaire LL(1) que nous avons prouvé engendrer exactement le même langage que la grammaire normalisée d'ASN.1, et nous avons réalisé directement à partir de celle-ci un analyseur syntaxique en OCaml, établissant ainsi sa correction et sa complétude par rapport à la norme (en BNF). Même les macros, extensions dynamiques de la syntaxe de base, ont été traitées (avec certaines restrictions pour éviter l'écueil de l'indécidabilité) grâce au fait qu'OCaml est un langage fonctionnel permettant l'ordre supérieur.

Asno est une contribution de la recherche publique française à la communauté internationale des télécommunications, et en particulier sa distribution en mode source et sa gratuité dans un monde où les informations circulent difficilement à cause des enjeux économiques, tout cela

fait que cet outil a été très favorablement évalué, accueilli et réutilisé dans l'industrie (notamment une compagnie de téléphonie publique étasunienne : la Southwestern Bell Technology Inc.). Le Centre National de Recherche en Télécommunication de France Télécom (CNET) utilise notre ananlyseur syntaxique comme base de développement d'outils autour d'ASN.1, et notre formalisation y sert de base à des travaux de recherche actuellement.

Nous pensons avoir tenu nos objectifs de satisfaire à la fois la communauté des théoriciens des langages par une étude formelle et des preuves rigoureuses, et la communauté des télécommunicants par un outil logiciel qui est déjà plus qu'un prototype et qui peut servir de point de départ pour lancer un produit commercial de haute technologie.

Maintenant une piste de recherche intéressante concerne le codage. Les besoins d'utilisation optimale de la bande passante lors des transmissions (réseaux à hauts débits, vidéoconférence, multimédia, cables, satellites, etc.) a incité l'ISO et l'ITU-T à normaliser un codage compact des valeurs ASN.1, appelé *Packed Encoding Rules* (PER, X.691) (ITU, 1994f). À la différence de X.690 (BER), les PER ne transmettent pas ou très peu d'informations de contrôle, c'est-à-dire que les codes ne contiennent pas d'étiquetages, par exemple. Pour conserver une forme de contrôle sémantique des types (qui implique un décodage qui se fonderait sur lui, c'est-à-dire que le succès du contrôle implique celui du décodage), un certain nombre de vérifications et d'ordres canoniques sont imposés aux spécifications ASN.1, qui sont alors en hypothèse (en contexte) pour les deux communicants, et leur permet de bien se comprendre (malgré le non-dit). Le problème est que cette norme X.691 se présente comme une somme d'optimisations obscures, et il est très difficile d'en comprendre l'architecture. Une formalisation de tout ou partie des PER pour définir un autre codage de référence, et ensuite prouver à nouveau nos théorèmes serait une étape importante dans la conciliation de la sûreté des transmissions et leur rapidité.

Quant à la portée de nos résultats théoriques (correction et déterminisme), nous avons montré informellement à la section 3.9 page 59 que notre réécriture d'ASN.1 :1994 vers un noyau plus simple (sur lequel s'appuient nos théorèmes) est correcte, c'est-à-dire qu'elle préserve les codes de référence (formalisant ceux obtenus via les BER (ITU, 1994e)). Nous pensons de même que cette réécriture préserve aussi les codes PER (ITU, 1994f), ce qui constituerait un premier pas dans cette recherche future qui devrait englober les PER. Toujours concernant les PER, nous pensons que notre simplification des contraintes de sous-typage serait correcte vis-à-vis des PER, c'est-à-dire qu'elle préserverait les codes des valeurs des sous-types. Plus encore, puisque nous simplifions, nous fai-

sons apparaître des contraintes que les PER sont susceptibles d'employer (X.691 les qualifie de *PER-visible constraints*). Il nous semble tout à fait possible de poursuivre nos réécritures de façon à obtenir une réduction qui soit optimale pour les PER, c'est-à-dire qu'elle ferait apparaître le maximum de contraintes exploitables par les PER, et donc qui potentiellement ferait gagner du débit de transmission. Par exemple, nous nous arrêtons dans notre thèse à :

```
T ::= VisibleString (FROM ("A") | FROM ("A".."B"))
```

car cela nous suffit pour nos vérification sémantique. Mais il est possible dans ce cas de poursuivre la réduction, de façon à faire apparaître une contrainte PER-visible :

```
T ::= VisibleString (FROM ("A".."B"))
```

Cela suppose essentiellement de définir une inclusion entre contraintes pour simplifier les disjonctions. Un autre exemple serait la contrainte :

```
A ::= B (C EXCEPT D | C)
```

qui devrait être simplifié en

```
A ::= B (C)
```

pour devenir PER-visible.

Annexe

Chapitre 5

Formalisme

Dans ce chapitre, nous définissons ici une syntaxe abstraite pour ASN.1. Pour ce faire nous employons les définitions de types de Caml, et des références seront constamment faites à la recommandation X.680. Les noms de types et de valeurs Caml seront en anglais, comme il est de bon usage en informatique.

Il faut savoir que les constructeurs de valeurs Caml qui suivent ne sont pas tous produits par l'analyseur syntaxique : certains sont introduits par la phase de canonisation des types et des valeurs ASN.1, qui s'effectue sur l'arbre de syntaxe abstraite tel qu'il sort de l'analyseur syntaxique. Cette phase a pour but de désambigüer le plus possible ce qui peut l'être pour simplifier les traitements ultérieurs. Ainsi le fragment de syntaxe concrète `0` donne toujours en sortie de l'analyseur syntaxique l'arbre de syntaxe abstraite `Zero`, et la phase de canonisation produira soit l'arbre `0.0`, soit `Int`(0), selon le type de l'expression (cf. 6.7.1 page 132). À chaque étape de la présentation de la syntaxe abstraite, divisée en concepts comme les types, les valeurs, les étiquettes, etc., nous donnerons les conventions de notation des valeurs associées (par exemple, e dénote toujours un étiquetage, et donc est du type $\mathcal{E}$).

Puis les environnements, leurs opérations et notations propres seront présentés. Nous présenterons ensuite les conventions et le vocabulaire propre au formalisme (les propositions logiques et leurs preuves). Enfin, nous récapitulerons dans un glossaire le vocabulaire propre à cette thèse, qui aura été introduit dans ce chapitre.

5.1 Convention lexicales et typographiques

À la base, nous employons dans cette thèse les conventions lexicales du langage OCaml. Les principales déviances ou choix typographiques

sont :

- Les mots-clés apparaissent en **gras**.
- Les noms de types ou de valeurs sont dans la police du texte (par exemple, string, default_tagging), ou en fonte calligraphique (par exemple, $\mathcal{E}$).
- Les constructeurs de types et de valeurs apparaissent en police Sans Serif, par exemple list, PRIVATE.
- Des symboles ou des nombres peuvent intervenir dans la formation des noms de constructeurs de valeurs. ex. $\leqslant$, **..**, **{**, **}**, 0.0.
- Des espaces blanches sont autorisées dans les noms de constructeurs, par exemple, COMPONENTS OF, toujours dans le but de suggérer le plus possible le fragment de syntaxe concrète associé.
- Les constructeurs suivront la convention ASN.1 des mots-clés (tout en majuscules) lorsqu'ils dénotent le concept associé, par exemple, EXPLICIT. Sinon seul leur premier caractère sera en majuscule, par exemple, Context (pour les étiquettes contextuelles, qu'aucun mot-clé ne dénote).
- Les contructeurs de valeurs sont curryfiés et la position de leurs paramètres est indiquée dans la déclaration par un souligné, par exemple, Tag _ _. Les valeurs construites peuvent être ou non currifiées, comme Tag ψ IMPLICIT, ou Tag (ψ, IMPLICIT).
- Quand une notation de valeur ou de partie de valeur sera donnée, par exemple Φ, il faut entendre qu'elle peut aussi apparaître sous la forme primée (Φ'), surlignée ($\overline{\Phi}$), ou indicée (Φ_i). Généralement la première forme s'utilise pour signifier que le terme en question est un sous-élément d'une structure récursive (qui porte donc le même nom, mais sans prime), par exemple on écrira $\varphi' :: \Phi'$ pour une valeur qui s'appelle Φ. La seconde forme s'emploiera généralement pour signifier que le terme est inféré, c'est-à-dire qu'il est le résultat d'un calcul ou d'une réécriture à partir d'un terme de même nom, sans surlignage. La forme indicée quant à elle sous-entendra généralement que la valeur désignée est un terme dans un ensemble fini (ou une liste).
- Nous supposons dans cette thèse que nous disposons d'un type polymorphe prédéfini set, qui réalise un ensemble non ordonné. Nous le supposons défini concrètement par deux constructeurs (à l'instar de list) : {_} et {} (ou $\varnothing$).
- Les listes et ensembles seront parfois donnés en extension. Dans ce cas nous indicerons les constructeurs correspondants. Par exemple : $[\mathrm{T}]_{1 \leqslant i \leqslant n}$ est une liste non vide de n types T_i. De même $\{\mathrm{T}\}_{1 \leqslant i \leqslant n}$ est un ensemble non vide de n types T_i. Il est important de noter que cette notation indicée sert uniquement pour les ensembles

ou listes *non vides*. Nous utiliserons cette notation aussi comme filtre (il faudra donc considérer comme liée la variable n, dans nos exemples).

5.2 Étiquettes et étiquetage

Nous appelons *classe d'étiquette* toute valeur du type :

```
type class =
  UNIVERSAL
| APPLICATION
| PRIVATE
| Context
```

Cette définition est isomorphe à la syntaxe concrète donnée en (ITU, 1994a, § 28.1) (règle de grammaire `Class`). Pour l'usage de ces classes voir (ITU, 1994a, § 28.4) et (ITU, 1994a, § F.2.12). Les classes d'étiquettes seront notées c (cf. relation $\vdash_{\overline{8}}$ dans 6.10.3).

Nous nommons *numéro d'étiquette* toute valeur du type :

```
type tag_num =
  Imm of   int
| Ref of   string
```

Cette définition est isomorphe à la syntaxe concrète donnée en (ITU, 1994a, § 28.1) (règle de grammaire `ClassNumber`). Le constructeur Imm spécifie que le numéro est immédiat (par opposition à une référence, un lien, une indirection), et est un entier naturel. En réalité, la norme précise (ITU, 1994a, § 28.2) que l'entier en question doit être positif, mais le système de types de OCaml ne permet pas de dire cela, aussi cette vérification est laissée pour la canonisation des étiquettes (cf. relation $\vdash_{\overline{8}}$ à la section 6.10.3 page 141). Le constructeur Ref indique que le numéro d'étiquette est la valeur dont le nom suit (c'est donc une référence). Dans ce dernier cas, il faut consulter l'environnement des valeurs pour connaître le numéro.

Nous nommons *attribut d'étiquette* toute valeur du type :

```
type attribute =
  Inferred
| IMPLICIT
| EXPLICIT
```

Cette définition est isomorphe à la syntaxe concrète donnée en (ITU, 1994a, § 28.1) (règle de grammaire `TaggedType`). Le constructeur Inferred correspond au cas où l'attribut doit être inféré selon le contexte (mode d'étiquetage par défaut du module par exemple). La connaissance de ce contexte permet alors de réécrire ce constructeur en l'un ou l'autre des constructeurs qui suivent dans la définition. Pour la sémantique informelle détaillée de ces concepts voir (ITU, 1994a, § 28). Les attributs d'étiquettes seront notés a.

Nous appelons *mode d'étiquetage par défaut* la valeur :

type default_tags =
  EXPLICIT TAGS
| IMPLICIT TAGS
| AUTOMATIC TAGS
val default_tagging : default_tags option

La définition de default_tag est isomorphe au trois premières productions de la règle de grammaire `TagDefault`, en (ITU, 1994a, § 10.1). Les modes d'étiquetage seront toujours utilisés en combinaison avec le type polymorphe option. Ainsi None correspondra à la production vide de la règle de grammaire associée.

Nous nommons *identifiant d'étiquette* toute valeur du type :

type tag_id = class × tag_num

Les identifiants d'étiquettes sont toujours notés ψ dans cette thèse.

Nous nommons *étiquette* toute valeur du type :

type tag = Tag __ **of** tag_id × attribute

Les étiquettes sont notées t dans cette thèse.

Nous appelons *étiquetage* toute valeur du type :

type $\mathcal{E}$ = tag list

Les étiquetages sont notés τ dans cette thèse.

Pour fixer les idées, soit l'extrait de syntaxe concrète :

```
x INTEGER ::= 3
T ::= [x] EXPLICIT [PRIVATE 4] IMPLICIT BOOLEAN
```

Nous avons deux étiquettes : `[x] EXPLICIT` et `[4 PRIVATE] IMPLICIT`. La suite des deux constitue un étiquetage. La première étiquette produit l'arbre de syntaxe abstraite : Tag (Context, Ref ("x")) EXPLICIT, et la seconde : Tag (PRIVATE, Imm (4)) IMPLICIT. Nous avons l'étiquetage :

[Tag (Context, Ref ("x")) EXPLICIT; Tag (PRIVATE, Imm (4)) IMPLICIT]

5.3 Types

Nous définissons ici la syntaxe abstraite des types ASN.1. Le type OBJECT IDENTIFIER a été volontairement ignoré ici, son intérêt théorique étant nul. Les différents types de chaînes de caractères sont définis simplement par :

type strkind =
  Numeric
| Printable
| Teletex
| T_{61}
| Videotex
| IA_5
| Graphic
| Visible
| ISO_{646}
| General

Un *type* ASN.1 est une valeur du type OCaml ⊔ :

type $\mathcal{T}$ =

Toutes ses valeurs seront notées T. Voyons la liste de ses constructeurs, chacun correspondant à un type ASN.1.

CHOICE _ **of** field list

Ce constructeur représente le type somme `CHOICE` (ITU, 1994a, § 26). Il a pour paramètre une liste de variantes (voir plus bas). Cette liste de variantes sera notée $\mathcal{F}$. Par exemple :

```
T ::= CHOICE {
        a [0] INTEGER,
        b [1] BOOLEAN
      }
```

définit une union de deux variantes de labels `a` et `b`. Une valeur de ce type sera obligatoirement une valeur pour le type d'une de ces variantes. Ce type correspond au `union` du langage C.

| SET _ **of** element list

Ce constructeur correspond au type `SET` (ITU, 1994a, § 24). C'est un produit appliqué à une liste de champs qui est notée Φ. Par exemple :

```
T ::= SET {
        a [0] INTEGER,
        b [1] BOOLEAN
      }
```

définit une structure composée de deux champs dont les labels sont `a` et `b`. Chacun de ces champs possède un type et un étiquetage. Une valeur de ce type doit obligatoirement (sauf attribut de champ `OPTIONAL` ou `DEFAULT`) fournir une valeur pour chacun des types des champs. L'ordre des champs n'est absolument pas significatif pour l'ordre des valeurs corespondant aux champs. Ce type SET correspond au `struct` du langage C, ou encore au `record` de Pascal.

| SEQUENCE _ **of** element list

Ce constructeur correspond au type `SEQUENCE` (ITU, 1994a, § 22). Comme SET, il dénote un type produit, avec la différence que le produit cartésien n'est pas commutatif ici : les valeurs pour chaque champ devront être écrites dans l'ordre des champs (de la déclaration de type). Il s'applique à une liste de champs. Cette liste de champs sera notée Φ.

| SET OF _ _ _ **of** $\mathcal{E} \times \mathcal{T} \times \mathcal{S}$

Ce constructeur correspond au type `SET OF` qui dénote un ensemble non ordonné avec répétition de valeurs ASN.1 de même type (ITU, 1994a, § 25). Le type des éléments est le second paramètre. Le premier paramètre est l'étiquetage associé à ce type, et le dernier paramètre est une contrainte de sous-typage s'appliquant au type des éléments (voir plus bas). Par exemple :

```
T ::= SET OF [7] INTEGER (0|1)
```

définit un ensemble non ordonné avec répétition de nombres entiers égaux à 0 ou 1, et dont l'étiquetage du type (INTEGER) est `[7]`.

| SEQUENCE OF _ _ _ **of** $\mathcal{E} \times \mathcal{T} \times \mathcal{S}$

Ce constructeur correspond au type `SEQUENCE OF` qui dénote un ensemble ordonné avec répétition de valeurs ASN.1 de même type (ITU, 1994a, § 23). La signification des paramètres est la même que pour le constructeur SET OF. Par exemple :

```
T ::= SEQUENCE OF INTEGER (0|1)
```

est une définition de l'ensemble des représentations binaires.

| _ < __ **of** string × $\mathcal{E}$ × $\mathcal{T}$

Ce constructeur a pour nom le symbole < qui dénote dans la syntaxe concrète le *type sélection* (on dira simplement une *sélection*). Se référer à (ITU, 1994a, § 27). Il s'agit de la projection associée au type somme CHOICE. Le premier paramètre est le nom de la variante (nous dirons aussi « *label* de variante ») à projeter. Le troisième est le type à projeter et le second l'étiquetage de ce dernier. Nous noterons le premier argument du constructeur : l. Par exemple :

```
T := a < CHOICE {a INTEGER, b BOOLEAN}
```

implique

```
T ::= INTEGER
```

Le constructeur suivant est :

| INTEGER _ **of** (string × $\mathcal{V}$) list

Il s'agit du type `INTEGER` (ITU, 1994a, § 16). Il prend comme argument une liste de couples de la forme (nom, valeur). Chacun de ces couples définit une liaison locale au type, il s'agit donc d'un sous-environnement de valeurs. Nous verrons dans le chapitre consacré à la canonisation des environnements comment traiter ces constantes entières (en particulier, la section 6.5.2 page 129). Nous noterons la liste de ces liaisons $\mathcal{C}$, et leur nom c. Prenons un exemple :

```
T ::= INTEGER {zero(0), mystic(7)}
```

alors nous pouvons définir la valeur `v` telle que :

```
v T ::= zero
```

Cette définition implique

```
v T ::= 0
```

Le constructeur suivant est :

| BIT STRING _ **of** (string × $\mathcal{V}$) list

Il s'agit du type `BIT STRING` (ITU, 1994a, § 19). Par rapport au constructeur précédent (INTEGER), les liaisons ici s'interprètent comme des positions nommées de bits dans une chaîne. Nous noterons la liste de ces liaisons $\mathcal{C}$, et leur nom c. Par exemple :

```
T ::= BIT STRING {lsb(0), msb(7)}
```

alors nous pouvons définir la chaîne de bits 10000000 de la façon suivante :

```
v T ::= {msb}
```

La présence des noms de position de bits s'interprète comme signifiant 1. Le constructeur suivant est :

| ENUMERATED _ **of** (string × $\mathcal{V}$) list

Il s'agit du type `ENUMERATED` (ITU, 1994a, § 17). Par rapport au constructeur INTEGER, les noms des liaisons ici dénotent l'ensemble des valeurs du type, les valeurs liées donnent au spécifieur le contrôle du codage des constantes énumérées. Nous noterons la liste de ces liaisons $\mathcal{C}$, et leur nom c. Par exemple :

```
Tao ::= ENUMERATED {yin(0), yang(1)}
```

alors le type `Tao` ne possède que deux valeurs : `yin` et `yang` :

```
moon Tao ::= yin
```

Les entiers associés à ces valeurs symboliques doivent êtres différents deux à deux, mais ne sont significatifs que pour le codage de la spécification. Ainsi :

```
moon Tao ::= 0
```

est *incorrect*. Le constructeur suivant est :

| TRef _ **of** string

Ce constructeur correspond aux `typereference` de la syntaxe concrète (ITU, 1994a, § 9.2), c'est-à-dire à ce que nous appellerons dans cette thèse *abréviation de type*. Le paramètre du constructeur est donc le nom d'un type présent dans un environnement qui doit être fourni en contexte. L'opération consistant à réécrire les abréviations de type en non abréviations s'appelera *dépliage de type*. Nous noterons l'argument de TRef : x. Par exemple, soit :

```
A ::= INTEGER
```

alors nous pouvons définir `B` de la façon suivante :

```
B ::= A
```

L'arbre de syntaxe abstraite définissant `B` est : TRef("A"). Les constructeurs suivants sont :

| BOOLEAN
| NULL
| OCTET STRING
| CharString _ **of** strkind

Le constructeur CharString introduit tous les types de chaînes de caractères. Dans cette thèse nous ne nous intéresserons pas aux alphabets qui distinguent ces types, car l'intérêt théorique est nul.

and element =
  Field _____ **of** field

Nous nommons *champ* toute valeur du type « element » construite grâce à Field. Nous noterons ces valeurs φ.

| COMPONENTS OF ___ **of** $\mathcal{E} \times \mathcal{T} \times \mathcal{S}$

Nous nommons *inclusion de champs* toute valeur du type « element » construite grâce à COMPONENTS OF. Pour le sens informel de cette construction se référer à (ITU, 1994a, § 22.4).

and field = string $\times$ $\mathcal{E}$ $\times$ $\mathcal{T}$ $\times$ $\mathcal{S}$ $\times$ status option

Ce type sert à définir les variantes (cf. CHOICE) et les champs (cf. SET, SEQUENCE et le type « element »). Une variante n'est pas autre chose qu'un champ de CHOICE. Un champ est donc une abréviation OCaml mise pour un quintuplet :

- le premier paramètre est le nom du champ, dit *label* et noté l,
- le troisième paramètre est le type du champ,
- le second paramètre est l'étiquetage du type du champ,
- le quatrième paramètre est la contrainte de sous-typage du type du champ,
- le cinquième paramètre est le *statut du champ* (voir ci-après), noté s.

Nous noterons les valeurs du type « field » : f. Prenons un exemple :

```
T ::= CHOICE {
        a INTEGER,
        b BOOLEAN
      }
```

Nous avons l'arbre de syntaxe abstraite :

CHOICE [("a", [], INTEGER [], {}, Some OPTIONAL);
("b", [], BOOLEAN, {}, Some OPTIONAL)]

On remarquera que les variantes des CHOICE ont un statut toujours égal à Some OPTIONAL pour des raisons d'uniformité dans le traitement avec les types SET et SEQUENCE.

and status =
  OPTIONAL
| DEFAULT _ **of** $\mathcal{V}$

Le type « status » sert à définir le *statut de champ*, en tant qu'argument du type OCaml option. Le premier constructeur (OPTIONAL) indique que le champ est optionnel, le second indique une valeur par défaut pour le champ (ITU, 1994a, § 22.8,§ 22.9).

and ...

5.4 Contraintes de sous-typage

Dans cette thèse nous n'aborderons pas les constructions ALL et EXCEPT (ITU, 1994a, § 44.1, § 44.2).

La particularité du sous-typage d'ASN.1 est qu'il est explicitement fondé sur le paradigme : « type = ensemble de valeurs ». Cela est dû au contexte télécom qui est son berceau et où l'aspect transmission de données guide l'intuition.

and ...

Nous appellerons *contrainte de sous-typage* toute valeur du type :

and $\mathcal{S}$ = base_and_intersection set (* Union *)

Une contrainte de sous-typage est une union (un ensemble non ordonné) de contraintes de base et/ou d'une intersection de contraintes.

Nous avons choisi de représenter les contraintes comme des unions d'intersections, plutôt que comme des intersections d'unions — ce qui correspond directement à la syntaxe concrète. La raison de ce choix apparaîtra au chapitre 9 page 217 consacré au contrôle des sous-types. Nous noterons les contraintes σ.

Nous présentons maintenant les contraintes de bases et les intersections de contraintes. Ce sont les valeurs du type :

and base_and_intersection =

Toutes ses valeurs seront notées ν. Le premier constructeur est :

Inter _ **of** $\mathcal{S}$ list (* Intersection *)

Le constructeur Inter introduit l'intersection de contraintes et consiste en une liste de contraintes. Ces intersections sont non commutatives en général. Nous noterons les valeurs du type « $\mathcal{S}$ list » : Σ.

(* Basic constraints *)
| Value _ **of** $\mathcal{V}$

Le constructeur Value correspond à (ITU, 1994a, § 45.2). Nous nommerons *contrainte de valeur* toute valeur du type « base_and_intersection » construite à l'aide de Value. Il s'agit de la contrainte qui spécifie en un point la valeur du type auquel elle s'applique et le restreint à cette unique valeur. Par exemple la contrainte du type INTEGER dans

```
T ::= INTEGER (7)
```

est $\{\mathsf{Value}\,(\mathsf{Int}\,(7))\}$, et

```
v T ::= 3
```

est *incorrect*.

| INCLUDES ___ **of** $\mathcal{E} \times \mathcal{T} \times \mathcal{S}$

Le constructeur INCLUDES correspond à (ITU, 1994a, § 45.3). Une *inclusion de contraintes* est une valeur du type « base_and_intersection » construite à l'aide de INCLUDES. Il s'agit de la contrainte qui spécifie qu'elle doit être remplacée par la contrainte de sous-typage du type en second argument, auquel s'applique déjà la contrainte en troisième argument. Le premier paramètre est l'étiquetage du type en second paramètre (voir example juste après). Le type dont les contraintes sont incluses et celui auquel s'applique la contrainte INCLUDES doivent être obtenus par sous-typage à partir du même type (qu'on nommera *sur-type*) :

```
Day ::= ENUMERATED {monday (0), tuesday (1), wednesday (2),
                    thursday (3), friday (4), saturday (5),
                    sunday (6)}
Week-End ::= Day (saturday | sunday)
Long-Week-End ::= Day (INCLUDES Week-End | monday)
```

La sémantique de INCLUDES implique que `Long-Week-End` doit être réécrit en

```
Long-Week-End ::= Day (saturday | sunday | monday)
```

Ici, `Week-End` et `Day` ont le même sur-type `Day`. Nous appellerons *dérivation* l'opération qui consiste à obtenir un type par sous-typage. Pour `Day` la longueur de dérivation est nulle (le sous-typage n'est pas strict en ASN.1, c'est-à-dire qu'un sous-type n'est pas de cardinal strictement inférieur à celui de son sur-type.) Le constructeur suivant est :

| __ **..** __ **of** bound × inout × inout × bound

Le constructeur **..** correspond à (ITU, 1994a, § 45.4). Une *contrainte d'intervalle* est une valeur du type « base_and_intersection » construite à l'aide de « **..** ». Cette contrainte spécifie un intervalle de valeurs pour le type auquel elle s'applique et qui se prête à une telle topologie : INTEGER, REAL et CharString. Les quatres paramètres sont deux paires mises à plat et constituées chacune d'une borne (cf. plus bas) et d'une indication pour savoir si la borne est incluse ou non. Par exemple la contrainte du type INTEGER dans

```
T ::= INTEGER (2 < .. 5)
```

correspond à la syntaxe abstraite : $(\mathsf{Val}\,(\mathsf{Int}\,(2))) < \mathbf{..} \leqslant (\mathsf{Val}\,(\mathsf{Int}\,(5)))$, soit, en notation mathématique : $]2;5]$. Le constructeur suivant est :

| SIZE _ **of** $\mathcal{S}$

Le constructeur SIZE est défini par la norme à la cote (ITU, 1994a, § 45.5). Une *contrainte de taille* est une valeur du type « base_and_intersection » construite à l'aide de SIZE. Cette contrainte spécifie le cardinal du type auquel elle s'applique : uniquement BIT STRING, OCTET STRING, CharString, SEQUENCE OF et SET OF. Le paramètre est une contrainte au sens général et nous devrons vérifier que cette contrainte est bien un cardinal. Par exemple,

```
T ::= SET SIZE (3) OF INTEGER
```

définit un ensemble de trois entiers. L'arbre de syntaxe abstraite associé au type T est $\mathsf{SET\ OF}\,[\,]\,(\mathsf{INTEGER}\,[\,])\,[\,]$, et celui associé à la contrainte

est $\{\mathsf{SIZE}\,(\{\mathsf{Value}\,(\mathsf{Int}\,(3))\})\}$. Le constructeur suivant est :

| FROM _ **of** $\mathcal{S}$

Le constructeur FROM correspond à la cote (ITU, 1994a, § 45.7). Une *contrainte d'alphabet* est une valeur du type « base_and_intersection » construite à l'aide de FROM. Cette contrainte s'applique aux types de chaînes de caractères et spécifie leur alphabet. Le paramètre est une contrainte au sens le plus général et nous devrons vérifier qu'elle est bien un alphabet. Par exemple,

```
T ::= VisibleString (FROM ("A".."C"))
```

définit un sous-ensemble (un sous-type) de chaînes dont l'alphabet est composé des lettres A, B et C. L'arbre de syntaxe abstraite associé à la contrainte précédente est :

$$\{\mathsf{FROM}\,(\{(\mathsf{Val}\,(\mathsf{Str}\,(\text{"A"}))) \leqslant \mathbf{..} \leqslant (\mathsf{Val}\,(\mathsf{Str}\,(\text{"C"})))\})\}$$

Le constructeur suivant est :

| WITH COMPONENT _ **of** $\mathcal{S}$

Le constructeur WITH COMPONENT est défini par la norme à la cote (ITU, 1994a, § 45.8.3). Nous nommerons *contrainte interne simple* toute valeur du type « base_and_intersection » construite à l'aide de WITH COMPONENT. Une contrainte interne simple est par définition une *contrainte interne* (ITU, 1994a, § 45.8.1). Elle s'applique au type des éléments d'un SET OF ou d'un SEQUENCE OF (et non à ces types). Son paramètre est donc une contrainte générale. Par exemple,

```
T ::= SET (WITH COMPONENT (0|1)) OF INTEGER
```

définit l'ensemble des entiers valant 0 ou 1. L'arbre de syntaxe abstraite associé au type est $\mathsf{SET}\ []\ (\mathsf{INTEGER}\ [])\ []$ et celui associé à la contrainte est $\{\mathsf{WITH\ COMPONENT}\,\{\mathsf{Value}\,(\mathsf{Int}\,(0)); \mathsf{Value}\,(\mathsf{Int}\,(1))\}\}$. Cette définition du type `T` équivaut à

```
T ::= SET OF INTEGER (0|1)
```

Ici, l'arbre de syntaxe abstraite associé au type est :

$$\mathsf{SET}\ []\ (\mathsf{INTEGER}\ [])\ \{\mathsf{Value}\,(\mathsf{Int}\,(0)); \mathsf{Value}\,(\mathsf{Int}\,(1))\}$$

et celui associé à la contrainte est simplement $\{\}$. Le constructeur suivant est :

| WITH COMPONENTS **{_ _}of** kind × field_constraint list

Le constructeur WITH COMPONENTS correspond à (ITU, 1994a, § 45.8.4). Nous nommerons *contrainte interne multiple* toute valeur du type « base_and_intersection » construite à l'aide de WITH COMPONENTS. Une contrainte interne multiple est par définition une *contrainte interne* (ITU, 1994a, § 45.8.1). Elle s'applique aux types CHOICE, SET et SEQUENCE. Le premier paramètre indique de quelle sorte est cette contrainte (cf. ci-après) et le second est une liste de contraintes qui s'appliquent aux variantes ou champs des CHOICE, SET et SEQUENCE (cf. plus bas). Il est important de noter que, rigoureusement, le nom du constructeur WITH COMPONENTS contient les accolades ouvrantes et fermantes en gras (pour les distinguer des accolades du constructeur d'ensembles). Nous noterons les valeurs du type « field_constraint list » : $\mathcal{K}$.

and kind =
  Partial
| Full

Les contraintes internes multiples peuvent être de deux sortes : *partielles* ou *complètes*. Nous noterons les valeurs du type « kind » m. Nous nommerons *borne* toute valeur du type :

and bound =
  MIN
| MAX
| Val _ **of** $\mathcal{V}$

Les deux premiers constructeurs indiquent que la borne est l'élément minimum ou maximum du type (qui doit posséder de tels éléments). Le troisième constructeur introduit une valeur comme borne. Nous noterons les bornes e.

and inout = $<$ |$\leqslant$

Le type « inout » spécifie si la borne est exclue ou non de l'intervalle. Nous noterons les valeurs de ce type b. Nous nommerons *contrainte de champ* toute valeur du type :

and field_constraint = string × $\mathcal{S}$ × presence option

Les contraintes de champs structurent les contraintes internes mul-

tiples (cf. WITH COMPONENTS plus haut) et s'appliquent aux champs du type auxquel les contraintes multiples englobantes s'appliquent. Le premier paramètre est le label (i.e. le nom du champ). Le second est la contrainte du champ dont le label précède. Le troisième est une contrainte de présence qui s'applique au champ dont le premier paramètre donne le nom. Nous noterons respectivmeent les trois paramètres : l, σ, $\hat{\pi}$. Les contraintes de champs seront notées k. Par exemple

```
T ::= SET {a INTEGER} (WITH COMPONENTS {a (0|1)})
```

Ici, l'arbre de syntaxe abstraite du type est SET [] (INTEGER []) {}. L'arbre de syntaxe abstraite de la contrainte est :

{WITH COMPONENTS **{**(Full, [("a",
{Value (Int (0)); Value (Int (1))},
Some PRESENT)])**}**}

La définition précédente équivaut à :

```
T ::= SET {a INTEGER (0|1)}
```

Ici, l'arbre de synatxe abstraite associé au type est :

SET [] (INTEGER []) {Value (Int (0)); Value (Int (1))}

et celui associé à la contrainte est simplement {}.

and presence =
PRESENT
| ABSENT
| OPTIONAL

Les trois constructeurs du type « presence » correspondent aux trois constructions homonymes dans (ITU, 1994a, § 45.8.9). Ils contraignent le statut d'un champ.

and ...

5.5 Valeurs

Nous définissons ici la syntaxe abstraite des valeurs ASN.1. Certains des constructeurs qui suivent ne sont pas produits par l'analyseur syntaxique, mais par la phase de canonisation des environnements (voir la section 6 page 119). D'aucuns disparaîtront au cours de cette même phase. Nous le signalerons au fur et à mesure. Les valeurs seront génériquement

notées : v. La valeur qui correspondait au type ANY (ITU, 1990) a été ignorée.

and ...

and $\mathcal{V}$ =

NULL

Le constructeur NULL est employé à la fois pour dénoter le type `NULL` et son unique valeur `NULL`.

| Int _ **of** int

Il s'agit des valeurs du type INTEGER.

| Zero
| 0.0

Le premier constructeur, Zero, est produit par l'analyseur syntaxique en présence du fragment de syntaxe concrète : 0, qui peut être interprété comme l'entier (INTEGER) nul ou le réel (REAL) nul. La phase de canonisation des valeurs tranchera cette ambigüité, et supprimera ce constructeur de l'arbre de syntaxe abstraite. Dans le cas entier, il produira en lieu et place : Int (0), et dans le cas réel : 0.0. Par exemple la sortie de l'analyseur syntaxique pour les valeurs suivantes

```
x INTEGER ::= 0
y REAL ::= 0
```

est Zero dans les deux cas. Après canonisation, `x` se verra attribuer Int (0) et `y` 0.0.

| TRUE
| FALSE

Il s'agit des deux valeurs du type BOOLEAN.

| PLUS-INFINITY
| MINUS-INFINITY

Ce sont deux des trois valeurs spéciales du type REAL (ITU, 1994a, § 18.6).

| Str _ **of** string

Ce constructeur Str représente toutes les valeurs des différents types de chaînes de caractères.

| HexStr _ **of** string
| BinStr _ **of** string

Le premier constructeur, HexStr, est produit par l'analyseur syntaxique en présence d'une chaîne numérique hexadécimale. La phase de normalisation des valeurs convertira les valeurs de ce genre en représentation binaire et les remplacera par la construction BinStr.

| Id _ **of** string
| Enum _ **of** string
| VRef _ **of** string

Les constructeurs Id, Enum et VRef sont étroitement liés. Le premier est produit par l'analyseur syntaxique en présence d'une `valuereference` ou d'un `identifier`, car ils sont lexicalement indistinguables et le contexte syntaxique ne permet généralement pas de trancher. La phase de canonisation des valeurs désambigüera ces cas et produira tantôt Enum s'il se révèle que Id désignait une constante énumérée, et tantôt VRef s'il s'agissait d'une abréviation de valeur (`valuereference`). Donc Id disparaît après cette phase. Par exemple, les arbres de syntaxe abstraite pour les deux valeurs `x` et `y` suivantes :

```
T ::= ENUMERATED {a (0)}
x T ::= a
y T ::= x
```

sont respectivement : Id ("a") et Id ("x"). Après la phase de canonisation ils deviennent : Enum ("a") et VRef ("x"). Le lecteur attentif se demandera ce qu'il advient de

```
x INTEGER {a (1)} ::= a
```

L'analyseur syntaxique produit Id ("a") et la phase de canonisation lui substitue : Int (1)

| _ **:** _ **of** string $\times \mathcal{V}$

Il s'agit des valeurs du type CHOICE. Le premier paramètre représente le label de la variante dans le type (qu'on notera l) et le second la valeur pour le type de la variante.

| **{_}** **of** (string option × $\mathcal{V}$) list

Le constructeur **{_}** dénote toutes les valeurs qui n'ont pas été désambigüées (comme certaines chaînes numériques) et dont la syntaxe concrète est comprise entre deux accolades. Elles sont constituées d'une liste de couples (parfois l'ordre que la liste implique ne sera pas pertinent, mais le cas général l'exige, comme pour les valeurs du type SEQUENCE OF). La seconde composante des couples est une valeur. La première peut désigner une absence (None correspondant à `empty` dans la syntaxe concrète) ou un label (noté l). Par exemple l'arbre de syntaxe abstraite de la valeur `v` suivante

```
v SEQUENCE OF INTEGER ::= {0,1,2}
```

est **{**[(None, Int (0)); (None, Int (1)); (None, Int (2))]**}**. Celui dans :

```
x SET {a INTEGER, b BOOLEAN} ::= {a 3, b TRUE}
```

est **{**[(Some "a", Int (3)); (Some "b", TRUE)]**}**.

5.6 Environnements

Nous définissons ici les environnements de types et de valeurs, ainsi que les notations afférentes. Nous appellerons *domaine* toute valeur du type

type domain = string set

Un domaine est un ensemble de noms (on les notera x). Lorsqu'il s'agira d'un environnement de types, nous noterons le domaine : $\mathcal{A}$. S'il s'agit d'un environnement de valeurs : $\mathcal{B}$. Nous noterons x les noms de types, et y les noms de valeur.

type constrained_type = ($\mathcal{T} \times \mathcal{S}$) set

Le type « constrained_type » nous sert à capturer la forme normale disjonctive d'un sous-type : les unions de contraintes seront développées par rapport au type comme autant de termes algébriques. Nous noterons les valeurs de ce type : « ct ». Nous nommerons *les alias* toute valeur du type :

type alias = string list

Les alias nous servent à garder trace des abréviations de types modulo sous-typage, après les dépliages en cascade. Ils nous serviront pour

répondre à la question : quand deux types ont-ils un sur-type en commun ? Nous noterons α les alias. Nous nommons *un alias* un élément de la liste. Prenons l'exemple :

```
A ::= B (0|1)
B ::= C
C ::= INTEGER
```

La phase de canonisation des environnements de types dépliera les abréviations de type ci-dessus :

```
A ::= INTEGER (0|1)
B ::= INTEGER
C ::= INTEGER
```

Pour le contrôle des sous-types il nous faut conserver trace de ces dépliages, et cela grâce aux alias. Ainsi, les alias de `A` sont ["B"; "C"], ceux de `B` sont ["C"], et ceux de `C` sont [].

Nous nommerons *environnement de types* toute valeur de type :

type type_env = domain $\times$ (string $\rightarrow$ (alias $\times$ $\mathcal{E}$ $\times$ constrained_type))

Un environnement de types est une paire dont le premier composant est un domaine et le second une fonction des noms vers le triplet : (alias, étiquetage, sous-type en forme développée). Nous les noterons : $\langle\mathcal{A}\rangle\Gamma$. Cette définition provient en partie de la syntaxe concrète. Par exemple :

```
T ::= INTEGER (0|1)
```

correspond initialement à l'environnement de types

$$(\{\text{"T"}\}, \textbf{function}\ \text{"T"} \rightarrow ([\,], [\,], \{(\mathsf{INTEGER}\,[\,], \{\mathsf{Value}\,(\mathsf{Int}\,(0)); \mathsf{Value}\,(\mathsf{Int}\,(1))\})\}))$$

Lorsque le sous-type sera développé, nous aurons :

$$(\{\text{"T"}\}, \textbf{function}\ \text{"T"} \rightarrow ([\,], [\,], \{(\mathsf{INTEGER}\,[\,], \{\mathsf{Value}\,(\mathsf{Int}\,(0))\}); (\mathsf{INTEGER}\,[\,], \{\mathsf{Value}\,(\mathsf{Int}\,(1))\})\}))$$

Nous nommerons *liaison de type* une valeur du type :

type type_binding = string $\times$ (alias $\times$ $\mathcal{E}$ $\times$ constrained_type)

Nous les noterons γ. Les noms de types seront notés x. Nous utiliserons aussi une notation spéciale, destinée à rappeler que la paire en question est une liaison : $x \mapsto (\alpha, \tau, \{(\mathrm{T}, \sigma)\})$ est défini par : $(x, \alpha, \tau, \{(\mathrm{T}, \sigma)\})$.

Nous nommerons *environnement de valeurs* toute valeur du type :

type value_env = domain × (string → ($\mathcal{E} \times \mathcal{T} \times \mathcal{S} \times \mathcal{V}$))

Un environnement de valeurs est une paire dont le premier composant est un domaine et le second une fonction des noms vers un quadruplet : (étiquetage, type, contrainte de sous-typage, valeur). Nous les noterons : $\langle\mathcal{B}\rangle\Delta$. Cette définition est inspirée de la syntaxe concrète. Par exemple :

```
x BOOLEAN ::= TRUE
```

produit l'environnement de valeurs :

$$(\{\text{"x"}\}, \textbf{function}\ \text{"x"} \rightarrow ([\,], \mathsf{BOOLEAN}, \{\}, \mathsf{TRUE})).$$

Nous nommerons *liaison de valeur* une valeur du type :

type value_binding = string × ($\mathcal{E} \times \mathcal{T} \times \mathcal{S} \times \mathcal{V}$)

Nous les noterons δ. Les noms de valeurs seront notés y. Nous utiliserons aussi une notation spéciale, destinée à rappeler que la paire en question est une liaison : $y \mapsto (\tau, \mathrm{T}, \sigma, v)$ est défini par : $(y, \tau, \mathrm{T}, \sigma, v)$.

Nous définissons une fonction qui étend les environnements :

let $\oplus$ (dom,env) (u, v) =
  $(\mathrm{dom} \cup \{u\}, \textbf{fun}\, u' \rightarrow \textbf{if}\, u = u'\, \textbf{then}\, v\, \textbf{else}\, \mathrm{env}\,(u'))$

Nous l'emploierons en notation infixe : $\langle\mathcal{B}\rangle\Delta \oplus y \mapsto (\tau, \mathrm{T}, \sigma, v)$.

5.7 Règles d'inférence et preuves

Nous définissons ici les notations relatives aux règles d'inférence et les preuves. Les relations logiques sont reconnaissables à l'emploi du *symbole de thèse* $\vdash$, ou avec des variantes comme $\Vdash$. Un *jugement* est la mise en relation de *termes* : à droite du symbole de thèse se trouve un terme : *la thèse*, et à sa gauche se trouve un autre terme : le *contexte*, dans lequel la thèse doit être interprétée. Pour désigner ces positions par rapport au symbole de thèse, nous parlerons de *champ* : champ de contexte, champ de thèse. Par exemple,

$$\langle\mathcal{A}\rangle\Gamma \vdash \mathrm{T}$$

désigne une relation qui prend deux termes : la thèse T, et le contexte $\langle\mathcal{A}\rangle\Gamma$. Implicitement un jugement est vrai ou faux, c'est-à-dire qu'il est interprété comme vrai si et seulement si les termes sont en relation, et faux dans le cas contraire. Ce sont alors des *prédicats*. En termes fonctionnels on dirait que les relations sont des fonctions qui prennent deux

arguments : la thèse et le contexte, et retournent `true` ou `false` (les fonctions sont pour nous les fonctions de OCaml). Lorsque nous voudrons spécifier des fonctions retournant des valeurs de type quelconque, nous complèterons les jugements par une partie droite, introduite par le symbole $\rightarrow$, ou sa variante $\Rightarrow$. À droite de ce symbole nous aurons un champ *résultat*. Par exemple :

$$\langle \mathcal{A} \rangle \Gamma \vdash \mathrm{T} \rightarrow \overline{\mathrm{T}}$$

dénote un jugement dont l'interprétation fonctionnelle est une application de fonction aux deux arguments $\langle \mathcal{A} \rangle \Gamma$ et T, et de résultat $\overline{\mathrm{T}}$. Les champs de contexte et de thèse sont dits d'*analyse*, et celui de résultat d'*inférence*. En effet, les deux premiers sont des champs qui sont des *projections*, c'est-à-dire un terme est décomposé en sous-termes, alors que le dernier est une *injection*, c'est-à-dire qu'un terme est construit, à l'aide de constructeurs et de fonctions. En particulier cela signifie que le procédé de *filtrage* (projection) ne s'applique qu'aux contextes et thèses, autrement dit on ne pourra pas distinguer les deux conclusions de deux règles en usant du champ résultat.

Nous n'entrerons pas ici dans les détails de la sémantique opérationnelle structurelle (ou sémantique naturelle). Nous indiquons les particularités de notation propre à cette thèse.

— Le symbole $\triangleq$ introduit une liaison dans l'environnement de la règle (i.e. l'ensemble des liaisons formées par projections ou injections). Il correspond à la construction **let ... in ...** de OCaml. Par exemple :

$$\varphi \triangleq \mathsf{Field}\,(f)$$

introduit une nouvelle liaison de nom φ et de contenu $\mathsf{Field}\,(f)$. Cette construction ne possède pas d'interprétation au sens propre (i.e. une valeur booléenne) nous la considérerons comme une prémisse et sera placée en tant que telle dans la règle (une autre solution aurait été une construction **where** placée sous la conclusion.)

— Le symbole $\lhd$ dénote la projection explicite, c'est-à-dire la construction **match ... with ...** de OCaml. À gauche du symbole se trouve un terme et à droite un motif (donc contenant des noms qui sont ainsi liés dans l'environnement de la règle). Par exemple :

$$s' \lhd \mathsf{DEFAULT}\, v'$$

dénote la projection (ou filtrage) du terme s' en un terme DEFAULT dont la valeur est v' (ce nom est alors introduit dans la

règle et peut être référencé par les autres prémisses et un éventuel champ résultat.) En OCaml, ce jugement est équivalent à l'expression :

$$\textbf{match}\ s'\ \textbf{with}\ \mathsf{DEFAULT}\ (v').$$

Rappelons pour conclure que les prémisses et les règles ne sont pas ordonnées. Les motifs ne sont pas linéaires, et deux occurrences de même nom dans l'environnement de la règle sont possibles. Dans les deux cas cela signifie qu'il faut s'assurer que les deux termes de même nom sont identiques (i.e. syntaxiquement égaux).

Chapitre 6

Environnements canoniques

6.1 Ajout des types REAL et INTEGER

$$\frac{\begin{array}{c}\varphi_1 \triangleq \mathsf{Field}\,(\text{"mantissa"}, [\,], \mathsf{INTEGER}\,[\,], \{\}, \mathsf{None}) \\ \varphi_2 \triangleq \mathsf{Field}\,(\text{"base"}, [\,], \mathsf{INTEGER}\,[\,], \{\mathsf{Value}\,(\mathsf{Int}\,2); \mathsf{Value}\,(\mathsf{Int}\,10)\}, \mathsf{None}) \\ \varphi_3 \triangleq \mathsf{Field}\,(\text{"exponent"}, [\,], \mathsf{INTEGER}\,[\,], \{\}, \mathsf{None}) \\ \tau \triangleq [\mathsf{Tag}\ (\mathsf{UNIVERSAL}, \mathsf{Imm}\,9)\ \mathsf{IMPLICIT}] \\ \gamma_0 \triangleq \text{"REAL"} \mapsto ([\,], \tau, \{(\mathsf{SEQUENCE}\,[\varphi_1; \varphi_2; \varphi_3], \{\})\}) \\ \gamma_1 \triangleq \text{"INTEGER"} \mapsto ([\,], [\,], \{(\mathsf{INTEGER}\ [\,], \{\})\})\end{array}}{\vdash_{25} \langle\mathcal{A}\rangle\Gamma \to (\langle\mathcal{A}\rangle\Gamma \oplus \gamma_0) \oplus \gamma_1}$$

6.2 Types bien fondés

$$\frac{\Gamma(x) \lhd (\alpha, \tau, \{(\mathrm{T}, \sigma)\}) \qquad \langle\mathcal{A}\rangle\Gamma \Vdash \mathrm{T} \qquad \Vdash_{\overline{3}} \langle\mathcal{A}\rangle\Gamma}{\Vdash_{\overline{3}} \langle\{x\} \uplus \mathcal{A}\rangle\Gamma} \qquad \Vdash_{\overline{3}} \langle\varnothing\rangle\Gamma$$

$$\frac{\langle\mathcal{A}\rangle\Gamma, [\,], [\,], [\,] \Vdash_{\overline{1}} \mathrm{T}}{\langle\mathcal{A}\rangle\Gamma \Vdash \mathrm{T}}$$

$$\frac{(\mathrm{T}, [\,]) \in \mathrm{H}_1}{\langle\mathcal{A}\rangle\Gamma, \mathrm{H}_1, \mathrm{H}_0, [\,] \Vdash_{\overline{1}} \mathrm{T}}$$

$$\frac{(\mathrm{T}, [\,]) \notin \mathrm{H}_0 \qquad x' \in \mathcal{A} \qquad \Gamma(x') \lhd (\alpha', \tau', \{(\mathrm{T}', \sigma')\}) \qquad \langle\mathcal{A}\rangle\Gamma, \mathrm{H}_1, (\mathrm{T}, [\,]) :: \mathrm{H}_0, [\,] \Vdash_{\overline{1}} \mathrm{T}'}{\langle\mathcal{A}\rangle\Gamma, \mathrm{H}_1, \mathrm{H}_0, [\,] \Vdash_{\overline{1}} \mathsf{TRef}\,(x')\,\mathbf{as}\,\mathrm{T}}$$

$$\frac{(\mathrm{T}, l ::\_) \notin \mathrm{H}_0 \qquad x' \in \mathcal{A} \qquad \Gamma(x') \lhd (\alpha', \tau', \{(\mathrm{T}', \sigma')\}) \qquad \langle\mathcal{A}\rangle\Gamma, \mathrm{H}_1, (\mathrm{T}, \Lambda) :: \mathrm{H}_0, \Lambda \Vdash_{\overline{1}} \mathrm{T}'}{\langle\mathcal{A}\rangle\Gamma, \mathrm{H}_1, \mathrm{H}_0, (l ::\_\ \mathbf{as}\ \Lambda) \Vdash_{\overline{1}} \mathsf{TRef}\,(x')\,\mathbf{as}\,\mathrm{T}}$$

$$\frac{(\mathrm{T}, [\,]) \notin \mathrm{H}_0 \qquad \langle\mathcal{A}\rangle\Gamma, \mathrm{H}_1, (\mathrm{T}, [\,]) :: \mathrm{H}_0, [l'] \Vdash_{\overline{1}} \mathrm{T}'}{\langle\mathcal{A}\rangle\Gamma, \mathrm{H}_1, \mathrm{H}_0, [\,] \Vdash_{\overline{1}} (l' < \tau'\,\mathrm{T}')\,\mathbf{as}\,\mathrm{T}}$$

$$\frac{(\mathrm{T}, l ::\_) \notin \mathrm{H}_0 \qquad \langle\mathcal{A}\rangle\Gamma, \mathrm{H}_1, (\mathrm{T}, \Lambda) :: \mathrm{H}_0, l' :: \Lambda \Vdash_{\overline{1}} \mathrm{T}'}{\langle\mathcal{A}\rangle\Gamma, \mathrm{H}_1, \mathrm{H}_0, (l ::\_\ \mathbf{as}\ \Lambda) \Vdash_{\overline{1}} (l' < \tau'\,\mathrm{T}')\,\mathbf{as}\,\mathrm{T}}$$

$$\frac{\langle\mathcal{A}\rangle\Gamma, \mathrm{H}_1, \mathrm{H}_0, \Lambda \Vdash_{\overline{1}} \mathrm{T}}{\langle\mathcal{A}\rangle\Gamma, \mathrm{H}_1, (\_, l ::\_) :: \mathrm{H}_0, (l ::\_\ \mathbf{as}\ \Lambda) \Vdash_{\overline{1}} \mathsf{CHOICE}\ \ \_\,\mathbf{as}\,\mathrm{T}}$$

$$\frac{f' \lhd (l, \tau', \mathrm{T}', \sigma', s') \qquad \langle\mathcal{A}\rangle\Gamma, \mathrm{H}_1, \mathrm{H}_0, \Lambda \Vdash_{\overline{1}} \mathrm{T}'}{\langle\mathcal{A}\rangle\Gamma, \mathrm{H}_1, \mathrm{H}_0, l :: \Lambda \Vdash_{\overline{1}} \mathsf{CHOICE}\,([f'] \sqcup \mathcal{F}')}$$

$$\frac{\begin{array}{c}\mathrm{T} \lhd \mathsf{CHOICE}\,[(l_i,\, \tau_i,\, \mathrm{T}_i,\, \sigma_i,\, s_i)]_{1 \leqslant i \leqslant n} \\ \forall i \in [1..n].\langle \mathcal{A} \rangle \Gamma, (\mathrm{T}, [\,]) :: \mathrm{H}_0 \mathbin{@} \mathrm{H}_1, [\,], [\,] \Vdash_{\overline{1}} \mathrm{T}_i \\ \exists j \in [1..n].\langle \mathcal{A} \rangle \Gamma, \mathrm{H}_1, (\mathrm{T}, [\,]) :: \mathrm{H}_0, [\,] \Vdash_{\overline{1}} \mathrm{T}_j\end{array}}{\langle \mathcal{A} \rangle \Gamma, \mathrm{H}_1, \mathrm{H}_0, [\,] \Vdash_{\overline{1}} \mathrm{T}}$$

$$\frac{\begin{array}{c}\mathrm{T} \lhd (\mathsf{SEQUENCE} \mid \mathsf{SET})\,[\varphi_i]_{1 \leqslant i \leqslant n} \\ (\mathrm{T}, [\,]) \notin \mathrm{H}_0 \qquad \forall i \in [1..n].\langle \mathcal{A} \rangle \Gamma, \mathrm{H}_1, (\mathrm{T}, [\,]) :: \mathrm{H}_0, [\,] \Vdash_{\overline{2}} \varphi_i\end{array}}{\langle \mathcal{A} \rangle \Gamma, \mathrm{H}_1, \mathrm{H}_0, [\,] \Vdash_{\overline{1}} \mathrm{T}}$$

$$\frac{\begin{array}{c}\mathrm{T} \lhd \mathsf{INTEGER}\ \_ \mid \mathsf{BIT\ STRING}\ \_ \mid \mathsf{ENUMERATED}\ \_ \\ \mid \mathsf{BOOLEAN} \mid \mathsf{NULL} \mid \mathsf{OCTET\ STRING} \mid \mathsf{CharString}\ \_ \\ \mid \mathsf{SET\ OF}\ \_\_\_ \mid \mathsf{SEQUENCE\ OF}\ \_\_\_ \\ \mid \mathsf{SET}\ [\,] \mid \mathsf{SEQUENCE}\ [\,]\end{array}}{\langle \mathcal{A} \rangle \Gamma, \mathrm{H}_1, \mathrm{H}_0, [\,] \Vdash_{\overline{1}} \mathrm{T}}$$

$$\frac{s' \lhd \mathsf{Some}\,\mathsf{OPTIONAL}}{\langle \mathcal{A} \rangle \Gamma, \mathrm{H}_1, \mathrm{H}_0, [\,] \Vdash_{\overline{2}} \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s')}$$

$$\frac{s' \lhd \mathsf{Some}\,(\mathsf{DEFAULT}\ \_) \qquad \langle \mathcal{A} \rangle \Gamma, \mathrm{H}_1, \mathrm{H}_0, [\,] \Vdash_{\overline{1}} \mathrm{T}'}{\langle \mathcal{A} \rangle \Gamma, \mathrm{H}_1, \mathrm{H}_0, [\,] \Vdash_{\overline{2}} \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s')}$$

$$\frac{\langle \mathcal{A} \rangle \Gamma, \mathrm{H}_1, \mathrm{H}_0, [\,] \Vdash_{\overline{1}} \mathrm{T}'}{\langle \mathcal{A} \rangle \Gamma, \mathrm{H}_1, \mathrm{H}_0, [\,] \Vdash_{\overline{2}} \mathsf{COMPONENTS\ OF}\ \tau'\ \mathrm{T}'\ \sigma'}$$

6.3 Découplage des valeurs globales

$$\frac{\begin{array}{c}\Delta(y) \lhd ([\,], \mathsf{TRef}\ \_, \{\}, \_) \\ \vdash_{\overline{10}} (\langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta) \to (\langle\overline{\mathcal{A}}\rangle\overline{\Gamma}, \langle\overline{\mathcal{B}}\rangle\overline{\Delta}) \qquad \overline{\delta} \triangleq y \mapsto \Delta(y)\end{array}}{\vdash_{\overline{10}} (\langle\mathcal{A}\rangle\Gamma, \langle\{y\} \uplus \mathcal{B}\rangle\Delta) \to (\langle\overline{\mathcal{A}}\rangle\overline{\Gamma}, \langle\overline{\mathcal{B}}\rangle\overline{\Delta} \oplus \overline{\delta})}$$

$$\frac{\begin{array}{c}\Delta(y) \not\lhd ([\,], \mathsf{TRef}\ \_, \{\}, \_) \\ \Delta(y) \lhd (\tau, \mathrm{T}, \sigma, v) \qquad \vdash_{\overline{10}} (\langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta) \to (\langle\overline{\mathcal{A}}\rangle\overline{\Gamma}, \langle\overline{\mathcal{B}}\rangle\overline{\Delta}) \\ x \notin \overline{\mathcal{A}} \qquad \overline{\gamma} \triangleq x \mapsto ([\,], \tau, \{(\mathrm{T}, \sigma)\}) \qquad \overline{\delta} \triangleq y \mapsto ([\,], \mathsf{TRef}\,(x), \{\}, v)\end{array}}{\vdash_{\overline{10}} (\langle\mathcal{A}\rangle\Gamma, \langle\{y\} \uplus \mathcal{B}\rangle\Delta) \to (\langle\overline{\mathcal{A}}\rangle\overline{\Gamma} \oplus \overline{\gamma}, \langle\overline{\mathcal{B}}\rangle\overline{\Delta} \oplus \overline{\delta})}$$

$$\vdash_{\overline{10}} (\langle\mathcal{A}\rangle\Gamma, \langle\varnothing\rangle\Delta) \to (\langle\mathcal{A}\rangle\Gamma, \langle\varnothing\rangle\Delta)$$

6.4 Normalisation des types

6.4.1 Globalisation des types locaux sans leur étiquetage

$$\frac{\begin{array}{c}\Gamma(x) \lhd (\alpha, \tau, \{(\mathrm{T}_k, \sigma_k)\}_{1 \leqslant k \leqslant q}) \\ \{\}, (\{x\} \cup \mathcal{A})\Gamma \vdash_{\overline{37}} \{\mathrm{T}_k\}_{1 \leqslant k \leqslant q} \to (\{\overline{\mathrm{T}}_k\}_{1 \leqslant k \leqslant q}, \langle\mathcal{A}_0\rangle\Gamma_0) \\ \vdash_{\overline{65}} \langle\mathcal{A}_0 \setminus \{x\}\rangle\Gamma_0 \to \langle\overline{\mathcal{A}}_1\rangle\overline{\Gamma}_1\end{array}}{\vdash_{\overline{65}} \langle\{x\} \uplus \mathcal{A}\rangle\Gamma \to \langle\overline{\mathcal{A}}_1\rangle\overline{\Gamma}_1 \oplus x \mapsto (\alpha, \tau, \{(\overline{\mathrm{T}}_k, \sigma_k)\}_{1 \leqslant k \leqslant q})}$$

$$\vdash_{\overline{65}} \langle\varnothing\rangle\Gamma \to \langle\varnothing\rangle\Gamma$$

$$\frac{\langle\mathcal{A}\rangle\Gamma \vdash_{\overline{63}} \mathrm{T} \to (\overline{\mathrm{T}}, \langle\overline{\mathcal{A}}\rangle\overline{\Gamma}) \qquad \{\overline{\mathrm{T}}\} \cup \overline{\mathcal{T}}, \langle\overline{\mathcal{A}}\rangle\overline{\Gamma} \vdash_{\overline{37}} \mathcal{T} \to r}{\overline{\mathcal{T}}, \langle\mathcal{A}\rangle\Gamma \vdash_{\overline{37}} \{\mathrm{T}\} \uplus \mathcal{T} \to r}$$

$$\overline{\mathcal{T}}, \langle\mathcal{A}\rangle\Gamma \vdash_{\overline{37}} \{\} \to (\overline{\mathcal{T}}, \langle\mathcal{A}\rangle\Gamma)$$

$$\frac{x \notin \mathcal{A} \qquad \overline{\mathrm{T}} \triangleq l' < \tau'\,(\mathsf{TRef}\,x) \qquad \overline{\gamma} \triangleq x \mapsto ([\,],[\,],\{(\mathrm{T}',\{\})\})}{\langle\mathcal{A}\rangle\Gamma \vdash_{\overline{63}} l' < \tau'\,\mathrm{T}' \to (\overline{\mathrm{T}}, \langle\mathcal{A}\rangle\Gamma \oplus \overline{\gamma})}$$

$$\frac{x \notin \mathcal{A} \qquad \overline{\mathrm{T}} \triangleq \mathsf{SET\ OF}\ \tau'\,(\mathsf{TRef}\,x)\,\{\} \qquad \overline{\gamma} \triangleq x \mapsto ([\,],[\,],\{(\mathrm{T}',\sigma')\})}{\langle\mathcal{A}\rangle\Gamma \vdash_{\overline{63}} \mathsf{SET\ OF}\,\tau'\,\mathrm{T}'\,\sigma' \to (\overline{\mathrm{T}}, \langle\mathcal{A}\rangle\Gamma \oplus \overline{\gamma})}$$

$$\frac{\begin{array}{c}x \notin \mathcal{A} \\ \overline{\mathrm{T}} \triangleq \mathsf{SEQUENCE\ OF}\ \tau'\,(\mathsf{TRef}\,x)\,\{\} \qquad \overline{\gamma} \triangleq x \mapsto ([\,],[\,],\{(\mathrm{T}',\sigma')\})\end{array}}{\langle\mathcal{A}\rangle\Gamma \vdash_{\overline{63}} \mathsf{SEQUENCE\ OF}\,\tau'\,\mathrm{T}'\,\sigma' \to (\overline{\mathrm{T}}, \langle\mathcal{A}\rangle\Gamma \oplus \overline{\gamma})}$$

$$\frac{\begin{array}{c}f' \lhd (l', \tau', \mathrm{T}', \sigma', s') \qquad \langle\mathcal{A}\rangle\Gamma \vdash_{\overline{63}} \mathsf{CHOICE}\,\mathcal{F}' \to (\mathsf{CHOICE}\,\overline{\mathcal{F}}', \langle\overline{\mathcal{A}}\rangle\overline{\Gamma}) \\ x \notin \overline{\mathcal{A}} \qquad \overline{f}' \triangleq (l', \tau', \mathsf{TRef}\,x, \{\}, s') \qquad \overline{\gamma} \triangleq x \mapsto ([\,],[\,],\{(\mathrm{T}',\sigma')\})\end{array}}{\langle\mathcal{A}\rangle\Gamma \vdash_{\overline{63}} \mathsf{CHOICE}\,(f' :: \mathcal{F}') \to (\mathsf{CHOICE}\,(\overline{f}' :: \overline{\mathcal{F}}'), \langle\overline{\mathcal{A}}\rangle\overline{\Gamma} \oplus \overline{\gamma})}$$

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \qquad \langle\mathcal{A}\rangle\Gamma \vdash_{\overline{63}} \mathsf{SET}\,\Phi' \to (\mathsf{SET}\,\overline{\Phi}', \langle\overline{\mathcal{A}}\rangle\overline{\Gamma}) \\ x \notin \overline{\mathcal{A}} \qquad \overline{\varphi}' \triangleq \mathsf{Field}\,(l', \tau', \mathsf{TRef}\,x, \{\}, s') \qquad \overline{\gamma} \triangleq x \mapsto ([\,],[\,],\{(\mathrm{T}',\sigma')\})\end{array}}{\langle\mathcal{A}\rangle\Gamma \vdash_{\overline{63}} \mathsf{SET}\,(\varphi' :: \Phi') \to (\mathsf{SET}\,(\overline{\varphi}' :: \overline{\Phi}'), \langle\overline{\mathcal{A}}\rangle\overline{\Gamma} \oplus \overline{\gamma})}$$

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{COMPONENTS\ OF}\,\tau'\,\mathrm{T}'\,\sigma' \\ \langle\mathcal{A}\rangle\Gamma \vdash_{\overline{63}} \mathsf{SET}\,\Phi' \to (\mathsf{SET}\,\overline{\Phi}', \langle\overline{\mathcal{A}}\rangle\overline{\Gamma}) \qquad x \notin \overline{\mathcal{A}} \\ \overline{\varphi}' \triangleq \mathsf{COMPONENTS\ OF}\,\tau'\,(\mathsf{TRef}\,x)\,\{\} \qquad \overline{\gamma} \triangleq x \mapsto ([\,],[\,],\{(\mathrm{T}',\sigma')\})\end{array}}{\langle\mathcal{A}\rangle\Gamma \vdash_{\overline{63}} \mathsf{SET}\,(\varphi' :: \Phi') \to (\mathsf{SET}\,(\overline{\varphi}' :: \overline{\Phi}'), \langle\overline{\mathcal{A}}\rangle\overline{\Gamma} \oplus \overline{\gamma})}$$

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \\ \langle\mathcal{A}\rangle\Gamma \vdash_{\overline{63}} \mathsf{SEQUENCE}\,\Phi' \to (\mathsf{SEQUENCE}\,\overline{\Phi}', \langle\overline{\mathcal{A}}\rangle\overline{\Gamma}) \\ x \notin \overline{\mathcal{A}} \qquad \overline{\varphi}' \triangleq \mathsf{Field}\,(l', \tau', \mathsf{TRef}\,x, \{\}, s') \qquad \overline{\gamma} \triangleq x \mapsto ([\,],[\,],\{(\mathrm{T}',\sigma')\})\end{array}}{\langle\mathcal{A}\rangle\Gamma \vdash_{\overline{63}} \mathsf{SEQUENCE}\,(\varphi' :: \Phi') \to (\mathsf{SEQUENCE}\,(\overline{\varphi}' :: \overline{\Phi}'), \langle\overline{\mathcal{A}}\rangle\overline{\Gamma} \oplus \overline{\gamma})}$$

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{COMPONENTS\ OF}\,\tau'\,\mathrm{T}'\,\sigma' \\ \langle\mathcal{A}\rangle\Gamma \vdash_{\overline{63}} \mathsf{SEQUENCE}\,\Phi' \to (\mathsf{SEQUENCE}\,\overline{\Phi}', \langle\overline{\mathcal{A}}\rangle\overline{\Gamma}) \qquad x \notin \overline{\mathcal{A}} \\ \overline{\varphi}' \triangleq \mathsf{COMPONENTS\ OF}\,\tau'\,(\mathsf{TRef}\,x)\,\{\} \qquad \overline{\gamma} \triangleq x \mapsto ([\,],[\,],\{(\mathrm{T}',\sigma')\})\end{array}}{\langle\mathcal{A}\rangle\Gamma \vdash_{\overline{63}} \mathsf{SEQUENCE}\,(\varphi' :: \Phi') \to (\mathsf{SEQUENCE}\,(\overline{\varphi}' :: \overline{\Phi}'), \langle\overline{\mathcal{A}}\rangle\overline{\Gamma} \oplus \overline{\gamma})}$$

$$\frac{\begin{array}{c}\mathrm{T} \not\lhd \mathsf{CHOICE\ \_} \mid \mathsf{SET\ \_} \mid \mathsf{SEQUENCE\ \_} \\ \mid \mathsf{SET\ OF\ \_\_\_} \mid \mathsf{SEQUENCE\ OF\ \_\_\_} \mid \_ < \_\_\end{array}}{\langle\mathcal{A}\rangle\Gamma \vdash_{\overline{63}} \mathrm{T} \to (\mathrm{T}, \langle\mathcal{A}\rangle\Gamma)}$$

6.4.2 Globalisation des types inclus

$$\frac{x \in \mathcal{A} \qquad \Gamma(x) \lhd (\alpha, \tau, \{(\mathrm{T}, \sigma)\}) \qquad \langle \mathcal{A} \rangle \Gamma \vdash_{\overline{15}} \sigma \to (\overline{\sigma}, \langle \mathcal{A}_0 \rangle \Gamma_0) \qquad \vdash_{\overline{0}} \langle \mathcal{A}_0 \setminus \{x\} \rangle \Gamma_0 \to \langle \overline{\mathcal{A}} \rangle \overline{\Gamma}}{\vdash_{\overline{0}} \langle \mathcal{A} \rangle \Gamma \to \langle \overline{\mathcal{A}} \rangle \overline{\Gamma} \oplus x \mapsto (\alpha, \tau, \{(\mathrm{T}, \overline{\sigma})\})}$$

$$\vdash_{\overline{0}} \langle \varnothing \rangle \Gamma \to \langle \varnothing \rangle \Gamma$$

$$\frac{\langle \mathcal{A} \rangle \Gamma \vdash_{\overline{5}} \nu \to (\overline{\nu}, \langle \mathcal{A}_0 \rangle \Gamma_0) \qquad \langle \mathcal{A}_0 \rangle \Gamma_0 \vdash_{\overline{15}} \sigma \to (\overline{\sigma}, \langle \mathcal{A}_1 \rangle \Gamma_1)}{\langle \mathcal{A} \rangle \Gamma \vdash_{\overline{15}} \{\nu\} \uplus \sigma \to (\{\overline{\nu}\} \cup \overline{\sigma}, \langle \mathcal{A}_1 \rangle \Gamma_1)}$$

$$\langle \mathcal{A} \rangle \Gamma \vdash_{\overline{15}} \{\} \to (\{\}, \langle \mathcal{A} \rangle \Gamma)$$

$$\frac{\langle \mathcal{A} \rangle \Gamma \vdash_{\overline{15}} \sigma \to (\overline{\sigma}, \langle \mathcal{A}_0 \rangle \Gamma_0) \qquad \langle \mathcal{A}_0 \rangle \Gamma_0 \vdash_{\overline{5}} \mathsf{Inter}\, \Sigma \to (\mathsf{Inter}\, \overline{\Sigma}, \langle \mathcal{A}_1 \rangle \Gamma_1)}{\langle \mathcal{A} \rangle \Gamma \vdash_{\overline{5}} \mathsf{Inter}\, (\sigma :: \Sigma) \to (\mathsf{Inter}\, (\overline{\sigma} :: \overline{\Sigma}), \langle \mathcal{A}_1 \rangle \Gamma_1)}$$

$$\langle \mathcal{A} \rangle \Gamma \vdash_{\overline{5}} \mathsf{Inter}\, [\,] \to (\mathsf{Inter}\, [\,], \langle \mathcal{A} \rangle \Gamma)$$

$$\frac{x \notin \mathcal{A} \qquad \overline{\nu} \triangleq \mathsf{INCLUDES}\ [\,]\ (\mathsf{TRef}\, x)\ \{\} \qquad \langle \mathcal{A} \rangle \Gamma \vdash_{\overline{63}} \mathrm{T}' \to (\overline{\mathrm{T}}', \langle \overline{\mathcal{A}} \rangle \overline{\Gamma}) \qquad \overline{\gamma} \triangleq x \mapsto ([\,], \tau', \{(\overline{\mathrm{T}}', \sigma')\})}{\langle \mathcal{A} \rangle \Gamma \vdash_{\overline{5}} \mathsf{INCLUDES}\ \tau'\, \mathrm{T}'\, \sigma' \to (\overline{\nu}, \langle \overline{\mathcal{A}} \rangle \overline{\Gamma} \oplus \overline{\gamma})}$$

$$\frac{\langle \mathcal{A} \rangle \Gamma \vdash_{\overline{15}} \sigma \to (\overline{\sigma}, \langle \mathcal{A}_0 \rangle \Gamma_0)}{\langle \mathcal{A} \rangle \Gamma \vdash_{\overline{5}} \mathsf{SIZE}\, \sigma \to (\mathsf{SIZE}\, \overline{\sigma}, \langle \mathcal{A}_0 \rangle \Gamma_0)}$$

$$\frac{\langle \mathcal{A} \rangle \Gamma \vdash_{\overline{15}} \sigma \to (\overline{\sigma}, \langle \mathcal{A}_0 \rangle \Gamma_0)}{\langle \mathcal{A} \rangle \Gamma \vdash_{\overline{5}} \mathsf{FROM}\, \sigma \to (\mathsf{FROM}\, \overline{\sigma}, \langle \mathcal{A}_0 \rangle \Gamma_0)}$$

$$\frac{\langle \mathcal{A} \rangle \Gamma \vdash_{\overline{15}} \sigma \to (\overline{\sigma}, \langle \mathcal{A}_0 \rangle \Gamma_0) \qquad \overline{\nu} \triangleq \mathsf{WITH\ COMPONENT}\, \overline{\sigma}}{\langle \mathcal{A} \rangle \Gamma \vdash_{\overline{5}} \mathsf{WITH\ COMPONENT}\, \sigma \to (\overline{\nu}, \langle \mathcal{A}_0 \rangle \Gamma_0)}$$

$$\frac{\mathcal{K} \lhd [(l_j, \sigma_j, \hat{\pi}_j)]_{1 \leqslant j \leqslant p} \qquad \{\}, \langle \mathcal{A} \rangle \Gamma \vdash_{\overline{39}} \{\sigma_j\}_{1 \leqslant j \leqslant p} \to (\{\overline{\sigma}_j\}_{1 \leqslant j \leqslant p}, \langle \overline{\mathcal{A}} \rangle \overline{\Gamma}) \quad \overline{\mathcal{K}} \triangleq [(l_j, \overline{\sigma}_j, \hat{\pi}_j)]_{1 \leqslant j \leqslant p} \qquad \overline{\nu} \triangleq \mathsf{WITH\ COMPONENTS}\,(m, \overline{\mathcal{K}})}{\langle \mathcal{A} \rangle \Gamma \vdash_{\overline{5}} \mathsf{WITH\ COMPONENTS}\,(m, \mathcal{K}) \to (\overline{\nu}, \langle \overline{\mathcal{A}} \rangle \overline{\Gamma})}$$

$$\frac{\langle \mathcal{A} \rangle \Gamma \vdash_{\overline{15}} \sigma \to (\overline{\sigma}, \langle \overline{\mathcal{A}} \rangle \overline{\Gamma}) \qquad \{\overline{\sigma}\} \cup \overline{\mathcal{S}}, \langle \overline{\mathcal{A}} \rangle \overline{\Gamma} \vdash_{\overline{39}} \mathcal{S} \to r}{\overline{\mathcal{S}}, \langle \mathcal{A} \rangle \Gamma \vdash_{\overline{39}} \{\sigma\} \uplus \mathcal{S} \to r}$$

$$\overline{\mathcal{S}}, \langle \mathcal{A} \rangle \Gamma \vdash_{\overline{39}} \{\} \to (\overline{\mathcal{S}}, \langle \mathcal{A} \rangle \Gamma)$$

6.4.3 Dépliage des abréviations et sélections globales

$$\frac{\langle \mathcal{A} \rangle \Gamma \vdash_{\overline{66}} \langle \mathcal{A} \rangle \Gamma \to \langle \overline{\mathcal{A}} \rangle \overline{\Gamma}}{\vdash_{\overline{68}} \langle \mathcal{A} \rangle \Gamma \to \langle \overline{\mathcal{A}} \rangle \overline{\Gamma}}$$

$$\frac{\Gamma(x) \lhd (\alpha, \tau, \mathrm{ct}) \qquad \langle \mathcal{A} \rangle \Gamma, [\,] \vdash_{\overline{82}} (\tau, \mathrm{ct}) \to (\overline{\alpha}, \overline{\tau}, \overline{\mathrm{ct}}) \quad \langle \mathcal{A} \rangle \Gamma \vdash_{\overline{66}} \langle \mathcal{A}' \rangle \Gamma \to \langle \overline{\mathcal{A}'} \rangle \overline{\Gamma} \qquad \overline{\gamma} \triangleq x \mapsto (\alpha \mathbin{@} \overline{\alpha}, \overline{\tau}, \overline{\mathrm{ct}})}{\langle \mathcal{A} \rangle \Gamma \vdash_{\overline{66}} \langle \{x\} \uplus \mathcal{A}' \rangle \Gamma \to \langle \overline{\mathcal{A}'} \rangle \overline{\Gamma} \oplus \overline{\gamma}}$$

$$\langle \mathcal{A} \rangle \Gamma \vdash_{\overline{66}} \langle \varnothing \rangle \to \langle \varnothing \rangle$$

$$\frac{\forall k \in [1..q]. \langle \mathcal{A} \rangle \Gamma, \Lambda \vdash_{\overline{64}} (\tau, \mathrm{T}_k, \sigma_k) \to (\overline{\alpha}, \overline{\tau}, \overline{\mathrm{ct}}_k)}{\langle \mathcal{A} \rangle \Gamma, \Lambda \vdash_{\overline{82}} (\tau, \{(\mathrm{T}_k, \sigma_k)\}_{1 \leqslant k \leqslant q}) \to (\overline{\alpha}, \overline{\tau}, \cup_{k=1}^{q} \overline{\mathrm{ct}}_k)}$$

$$\frac{\tau \not\triangleleft [\,] \ \vee \ \sigma \neq \{\} \qquad \langle \mathcal{A} \rangle \Gamma, \Lambda \vdash_{\overline{64}} ([\,], \mathrm{T}, \{\}) \to r}{\langle \mathcal{A} \rangle \Gamma, (\_ :: \_ \ \mathbf{as}\ \Lambda) \vdash_{\overline{64}} (\tau, \mathrm{T}, \sigma) \to r}$$

$$\frac{\begin{array}{c}\tau \not\triangleleft [\,] \ \vee \ \sigma \neq \{\} \\ \langle \mathcal{A} \rangle \Gamma, [\,] \vdash_{\overline{64}} ([\,], \mathrm{T}, \{\}) \to (\overline{\alpha}, \overline{\tau}, \{(\overline{\mathrm{T}}_k, \overline{\sigma}_k)\}_{1 \leqslant k \leqslant q})\end{array}}{\langle \mathcal{A} \rangle \Gamma, [\,] \vdash_{\overline{64}} (\tau, \mathrm{T}, \sigma) \to (\overline{\alpha}, \tau \mathbin{@} \overline{\tau}, \{(\overline{\mathrm{T}}_k, \{\mathsf{Inter}\,[\overline{\sigma}_k; \sigma]\})\}_{1 \leqslant k \leqslant q})}$$

$$\frac{x' \in \mathcal{A} \qquad \Gamma(x') \lhd (\alpha', \tau', \mathrm{ct}') \qquad \langle\mathcal{A}\rangle\Gamma, \Lambda \vdash_{\overline{82}} (\tau', \mathrm{ct}') \to (\overline{\alpha}, \overline{\tau}, \overline{\mathrm{ct}})}{\langle\mathcal{A}\rangle\Gamma, \Lambda \vdash_{\overline{64}} ([\,], \mathsf{TRef}\, x', \{\}) \to (x' :: \alpha' \mathbin{@} \overline{\alpha}, \overline{\tau}, \overline{\mathrm{ct}})}$$

$$\frac{\langle\mathcal{A}\rangle\Gamma, l :: \Lambda \vdash_{\overline{64}} ([\,], \mathrm{T}, \{\}) \to r}{\langle\mathcal{A}\rangle\Gamma, \Lambda \vdash_{\overline{64}} ([\,], l < \tau\, \mathrm{T}, \{\}) \to r}$$

$$\frac{f' \lhd (l, \tau', \mathrm{T}', \sigma', s') \qquad \langle\mathcal{A}\rangle\Gamma, \Lambda \vdash_{\overline{64}} (\tau', \mathrm{T}', \sigma') \to r}{\langle\mathcal{A}\rangle\Gamma, l :: \Lambda \vdash_{\overline{64}} ([\,], \mathsf{CHOICE}\,([f'] \sqcup \mathcal{F}'), \{\}) \to r}$$

$$\langle\mathcal{A}\rangle\Gamma, [\,] \vdash_{\overline{64}} ([\,], \mathrm{T}, \{\}) \to ([\,], \{(\mathrm{T}, \{\})\})$$

6.4.4 Résolution de AUTOMATIC TAGS et COMPONENTS OF

$$\frac{\begin{array}{c}\mathrm{default\_tagging} \lhd \mathsf{Some}\,(\mathsf{AUTOMATIC\ TAGS}) \\ \Gamma(x) \lhd (\alpha, \tau, \{(\mathrm{T}, \sigma)\}) \qquad \vdash_{\mathrm{auto}} \mathrm{T} \\ \langle\mathcal{A}\rangle\Gamma \vdash_{\mathrm{comp}} \mathrm{T} \to \mathrm{T}_0 \qquad 0 \vdash_{\mathrm{auto}} \mathrm{T}_0 \to \overline{\mathrm{T}} \qquad \vdash_{\overline{9}} \langle\mathcal{A}\rangle\Gamma \to \langle\overline{\mathcal{A}}\rangle\overline{\Gamma}\end{array}}{\vdash_{\overline{9}} \langle\{x\} \uplus \mathcal{A}\rangle\Gamma \to \langle\overline{\mathcal{A}}\rangle\overline{\Gamma} \oplus x \mapsto (\alpha, \tau, \{(\overline{\mathrm{T}}, \sigma)\})}$$

$$\frac{\begin{array}{c}\mathrm{default\_tagging} \not\lhd \mathsf{Some}\,(\mathsf{AUTOMATIC\ TAGS}) \\ \langle\mathcal{A}\rangle\Gamma \vdash_{\mathrm{comp}} \mathrm{T} \to \overline{\mathrm{T}} \qquad \vdash_{\overline{9}} \langle\mathcal{A}\rangle\Gamma \to \langle\overline{\mathcal{A}}\rangle\overline{\Gamma}\end{array}}{\vdash_{\overline{9}} \langle\{x\} \uplus \mathcal{A}\rangle\Gamma \to \langle\overline{\mathcal{A}}\rangle\overline{\Gamma} \oplus x \mapsto (\alpha, \tau, \{(\overline{\mathrm{T}}, \sigma)\})} \qquad \vdash_{\overline{9}} \langle\varnothing\rangle\Gamma \to \langle\varnothing\rangle\Gamma$$

$$\frac{\vdash_{\mathrm{auto}} \mathsf{SET}\, \Phi'}{\vdash_{\mathrm{auto}} \mathsf{SET}\,(\mathsf{Field}\,(l', [\,], \mathrm{T}', \sigma', s') :: \Phi')}$$

$$\frac{\vdash_{\mathrm{auto}} \mathsf{SET}\, \Phi'}{\vdash_{\mathrm{auto}} \mathsf{SET}\,((\mathsf{COMPONENTS\ OF}\, \tau'\, \mathrm{T}'\, \sigma') :: \Phi')}$$

$$\frac{\vdash_{\mathrm{auto}} \mathsf{SEQUENCE}\, \Phi'}{\vdash_{\mathrm{auto}} \mathsf{SEQUENCE}\,(\mathsf{Field}\,(l', [\,], \mathrm{T}', \sigma', s') :: \Phi')}$$

$$\frac{\vdash_{\mathrm{auto}} \mathsf{SEQUENCE}\, \Phi'}{\vdash_{\mathrm{auto}} \mathsf{SEQUENCE}\,((\mathsf{COMPONENTS\ OF}\, \tau'\, \mathrm{T}'\, \sigma') :: \Phi')}$$

$$\frac{\vdash_{\text{auto}} \mathsf{CHOICE}\,\mathcal{F}'}{\vdash_{\text{auto}} \mathsf{CHOICE}\,((l', [\,], \mathrm{T}', \sigma', s') :: \mathcal{F}')}$$

$$\frac{\mathrm{T} \not\lhd \mathsf{SET}\ \ \_ \mid \mathsf{SEQUENCE}\ \ \_ \mid \mathsf{CHOICE}\ \ \_}{\vdash_{\text{auto}} \mathrm{T}}$$

$$\frac{\begin{array}{c} n+1 \vdash_{\text{auto}} \mathsf{SET}\,\Phi' \to \mathsf{SET}\,\overline{\Phi}' \qquad \overline{\tau}' \triangleq (\mathsf{Tag}\ (\mathsf{Context}, n)\ \mathsf{Inferred}) :: \tau' \\ \overline{\mathrm{T}} \triangleq \mathsf{SET}\,(\mathsf{Field}\,(l', \overline{\tau}', \mathrm{T}', \sigma', s') :: \overline{\Phi}') \end{array}}{n \vdash_{\text{auto}} \mathsf{SET}\,(\mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') :: \Phi') \to \overline{\mathrm{T}}}$$

$$\frac{\begin{array}{c} n+1 \vdash_{\text{auto}} \mathsf{SEQUENCE}\,\Phi' \to \mathsf{SEQUENCE}\,\overline{\Phi}' \\ \overline{\tau}' \triangleq (\mathsf{Tag}\ (\mathsf{Context}, n)\ \mathsf{Inferred}) :: \tau' \\ \overline{\mathrm{T}} \triangleq \mathsf{SEQUENCE}\,(\mathsf{Field}\,(l', \overline{\tau}', \mathrm{T}', \sigma', s') :: \overline{\Phi}') \end{array}}{n \vdash_{\text{auto}} \mathsf{SEQUENCE}\,(\mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') :: \Phi') \to \overline{\mathrm{T}}}$$

$$\frac{\begin{array}{c} n+1 \vdash_{\text{auto}} \mathsf{CHOICE}\,\mathcal{F}' \to \mathsf{CHOICE}\,\overline{\mathcal{F}}' \\ \overline{\tau}' \triangleq (\mathsf{Tag}\ (\mathsf{Context}, n)\ \mathsf{Inferred}) :: \tau' \\ \overline{\mathrm{T}} \triangleq \mathsf{CHOICE}\,((l', \overline{\tau}', \mathrm{T}', \sigma', s') :: \overline{\mathcal{F}}') \end{array}}{n \vdash_{\text{auto}} \mathsf{CHOICE}\,((l', \tau', \mathrm{T}', \sigma', s') :: \mathcal{F}') \to \overline{\mathrm{T}}}$$

$$\frac{\mathrm{T} \not\lhd \mathsf{SET}\ \ \_ \mid \mathsf{SEQUENCE}\ \ \_ \mid \mathsf{CHOICE}\ \ \_}{n \vdash_{\text{auto}} \mathrm{T} \to \mathrm{T}}$$

$$\frac{\mathrm{T} \lhd \mathsf{SET}\ \ \_ \qquad \langle \mathcal{A} \rangle \Gamma \vdash_{26} \mathrm{T} \to \mathcal{F} \qquad \overline{\mathrm{T}} \triangleq \mathsf{SET}(\mathrm{List.map}\ \mathsf{Field}\ \mathcal{F})}{\langle \mathcal{A} \rangle \Gamma \vdash_{\text{comp}} \mathrm{T} \to \overline{\mathrm{T}}}$$

$$\frac{\begin{array}{c} \mathrm{T} \lhd \mathsf{SEQUENCE}\ \ \_ \\ \langle \mathcal{A} \rangle \Gamma \vdash_{26} \mathrm{T} \to \mathcal{F} \qquad \overline{\mathrm{T}} \triangleq \mathsf{SEQUENCE}\,(\mathrm{List.map}\ \mathsf{Field}\ \mathcal{F}) \end{array}}{\langle \mathcal{A} \rangle \Gamma \vdash_{\text{comp}} \mathrm{T} \to \overline{\mathrm{T}}}$$

$$\frac{\mathrm{T} \not\lhd \mathsf{SET}\ \ \_ \mid \mathsf{SEQUENCE}\ \ \_}{\langle \mathcal{A} \rangle \Gamma \vdash_{\text{comp}} \mathrm{T} \to \mathrm{T}}$$

Nous définissons une relation qui, étant donné un type, si c'est un SEQUENCE (liste finie ordonnée de types) ou un SET (liste finie non-ordonnée de types) qui spécifient des inclusions de champs, fournit la liste complète et expansée des champs, et sinon retourne la liste vide. Elle impose en plus que les SET ne peuvent inclure que des champs d'autres SET et idem pour les SEQUENCE.

$$\frac{\langle\mathcal{A}\rangle\Gamma \vdash_{26} \mathsf{SET}\,\Phi' \to \mathcal{F}'}{\langle\mathcal{A}\rangle\Gamma \vdash_{26} \mathsf{SET}\,(\mathsf{Field}\,(f')::\Phi') \to f'::\mathcal{F}'}$$

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{COMPONENTS\ OF}\,\tau'\,(\mathsf{TRef}\,x)\,\{\} \\ x \in \mathcal{A} \qquad \Gamma(x) \lhd (\alpha,\tau,\{(\mathsf{SET}\,\Phi,\sigma)\}) \\ \langle\mathcal{A}\rangle\Gamma \vdash_{26} \mathsf{SET}\,\Phi \to \mathcal{F}_0 \qquad \langle\mathcal{A}\rangle\Gamma \vdash_{26} \mathsf{SET}\,\Phi' \to \mathcal{F}_1\end{array}}{\langle\mathcal{A}\rangle\Gamma \vdash_{26} \mathsf{SET}\,(\varphi'::\Phi') \to \mathcal{F}_0 \mathbin{@} \mathcal{F}_1}$$

$$\frac{\langle\mathcal{A}\rangle\Gamma \vdash_{26} \mathsf{SEQUENCE}\,\Phi' \to \mathcal{F}'}{\langle\mathcal{A}\rangle\Gamma \vdash_{26} \mathsf{SEQUENCE}\,(\mathsf{Field}\,(f')::\Phi') \to f'::\mathcal{F}'}$$

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{COMPONENTS\ OF}\,\tau'\,(\mathsf{TRef}\,x)\,\{\} \\ x \in \mathcal{A} \qquad \Gamma(x) \lhd (\alpha,\tau,\{(\mathsf{SEQUENCE}\,\Phi,\sigma)\}) \\ \langle\mathcal{A}\rangle\Gamma \vdash_{26} \mathsf{SEQUENCE}\,\Phi \to \mathcal{F}_0 \qquad \langle\mathcal{A}\rangle\Gamma \vdash_{26} \mathsf{SEQUENCE}\,\Phi' \to \mathcal{F}_1\end{array}}{\langle\mathcal{A}\rangle\Gamma \vdash_{26} \mathsf{SEQUENCE}\,(\varphi'::\Phi') \to \mathcal{F}_0 \mathbin{@} \mathcal{F}_1}$$

$$\frac{\mathrm{T} \not\lhd \mathsf{SET}\ \_ \mid \mathsf{SEQUENCE}\ \_}{\langle\mathcal{A}\rangle\Gamma \vdash_{26} \mathrm{T} \to []}$$

6.4.5 Normalisation des types

$$\frac{\begin{array}{ccc}\vdash_{65} \langle\mathcal{A}\rangle\Gamma \to \langle\mathcal{A}_0\rangle\Gamma_0 & & \\ \vdash_{0} \langle\mathcal{A}_0\rangle\Gamma_0 \to \langle\mathcal{A}_1\rangle\Gamma_1 & \vdash_{68} \langle\mathcal{A}_1\rangle\Gamma_1 \to \langle\mathcal{A}_2\rangle\Gamma_2 & \vdash_{9} \langle\mathcal{A}_2\rangle\Gamma_2 \to \langle\mathcal{A}_3\rangle\Gamma_3\end{array}}{\vdash_{31} \langle\mathcal{A}\rangle\Gamma \to \langle\mathcal{A}_3\rangle\Gamma_3}$$

6.5 Découplage valeurs/types

6.5.1 Les valeurs par défaut

$$\frac{\begin{array}{c}\Gamma(x) \lhd (\alpha,\tau,\{(\mathrm{T},\sigma)\}) \\ \langle\mathcal{B}\rangle\Delta \vdash_{27} \mathrm{T} \to (\overline{\mathrm{T}},\langle\mathcal{B}_0\rangle\Delta_0) \qquad \langle\mathcal{B}_0\rangle\Delta_0 \vdash_{23} \langle\mathcal{A}\rangle\Gamma \to (\langle\overline{\mathcal{A}}\rangle\overline{\Gamma},\langle\mathcal{B}_1\rangle\Delta_1) \\ \langle\overline{\mathcal{A}}_0\rangle\overline{\Gamma}_0 \triangleq \langle\overline{\mathcal{A}}\rangle\overline{\Gamma} \oplus x \mapsto (\alpha,\tau,\{(\overline{\mathrm{T}},\sigma)\})\end{array}}{\langle\mathcal{B}\rangle\Delta \vdash_{23} \langle\{x\} \uplus \mathcal{A}\rangle\Gamma \to (\langle\overline{\mathcal{A}}_0\rangle\overline{\Gamma}_0,\langle\mathcal{B}_1\rangle\Delta_1)}$$

$$\langle\mathcal{B}\rangle\Delta \vdash_{23} \langle\varnothing\rangle\Gamma \to (\langle\varnothing\rangle\Gamma,\langle\mathcal{B}\rangle\Delta)$$

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\,(l', [\,], \mathsf{TRef}\,x, \{\}, \mathsf{Some}\,(\mathsf{DEFAULT}\,v')) \\ y \notin \mathcal{B} \qquad \langle\mathcal{B}_0\rangle\Delta_0 \triangleq \langle\mathcal{B}\rangle\Delta \oplus y \mapsto ([\,], \mathsf{TRef}\,x, \{\}, v') \\ \langle\mathcal{B}_0\rangle\Delta_0 \vdash_{27} \mathsf{SET}\,\Phi' \to (\mathsf{SET}\,\overline{\Phi}', \langle\mathcal{B}_1\rangle\Delta_1) \\ \overline{\varphi}' \triangleq \mathsf{Field}\,(l', [\,], \mathsf{TRef}\,x, \{\}, \mathsf{Some}\,(\mathsf{DEFAULT}\,(\mathsf{VRef}\,y)))\end{array}}{\langle\mathcal{B}\rangle\Delta \vdash_{27} \mathsf{SET}\,(\varphi' :: \Phi') \to (\mathsf{SET}\,(\overline{\varphi}' :: \overline{\Phi}'), \langle\mathcal{B}_1\rangle\Delta_1)}$$

$$\frac{\begin{array}{c}\varphi' \not\lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', \mathsf{Some}\,(\mathsf{DEFAULT}\,v')) \\ \langle\mathcal{B}\rangle\Delta \vdash_{27} \mathsf{SET}\,\Phi' \to (\mathsf{SET}\,\overline{\Phi}', \langle\mathcal{B}_0\rangle\Delta_0)\end{array}}{\langle\mathcal{B}\rangle\Delta \vdash_{27} \mathsf{SET}\,(\varphi' :: \Phi') \to (\mathsf{SET}\,(\varphi' :: \overline{\Phi}'), \langle\mathcal{B}_0\rangle\Delta_0)}$$

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\,(l', [\,], \mathsf{TRef}\,x, \{\}, \mathsf{Some}\,(\mathsf{DEFAULT}\,v')) \\ y \notin \mathcal{B} \qquad \langle\mathcal{B}_0\rangle\Delta_0 \triangleq \langle\mathcal{B}\rangle\Delta \oplus y \mapsto ([\,], \mathsf{TRef}\,x, \{\}, v') \\ \langle\mathcal{B}_0\rangle\Delta_0 \vdash_{27} \mathsf{SEQUENCE}\,\Phi' \to (\mathsf{SEQUENCE}\,\overline{\Phi}', \langle\mathcal{B}_1\rangle\Delta_1) \\ \overline{\varphi}' \triangleq \mathsf{Field}\,(l', [\,], \mathsf{TRef}\,x, \{\}, \mathsf{Some}\,(\mathsf{DEFAULT}\,(\mathsf{VRef}\,y)))\end{array}}{\langle\mathcal{B}\rangle\Delta \vdash_{27} \mathsf{SEQUENCE}\,(\varphi' :: \Phi') \to (\mathsf{SEQUENCE}\,(\overline{\varphi}' :: \overline{\Phi}'), \langle\mathcal{B}_1\rangle\Delta_1)}$$

$$\frac{\begin{array}{c}\varphi' \not\lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', \mathsf{Some}\,(\mathsf{DEFAULT}\,v')) \\ \langle\mathcal{B}\rangle\Delta \vdash_{27} \mathsf{SEQUENCE}\,\Phi' \to (\mathsf{SEQUENCE}\,\overline{\Phi}', \langle\mathcal{B}_0\rangle\Delta_0)\end{array}}{\langle\mathcal{B}\rangle\Delta \vdash_{27} \mathsf{SEQUENCE}\,(\varphi' :: \Phi') \to (\mathsf{SEQUENCE}\,(\varphi' :: \overline{\Phi}'), \langle\mathcal{B}_0\rangle\Delta_0)}$$

$$\frac{\mathrm{T} \not\lhd \mathsf{SET}\ \_ \mid \mathsf{SEQUENCE}\ \_}{\langle\mathcal{B}\rangle\Delta \vdash_{27} \mathrm{T} \to (\mathrm{T}, \langle\mathcal{B}\rangle\Delta)}$$

6.5.2 Les constantes des types

$$\frac{\begin{array}{c}\Gamma(x) \lhd (\alpha, \tau, \{(\mathrm{T}, \sigma)\}) \\ \langle\mathcal{B}\rangle\Delta \vdash_{29} \mathrm{T} \to (\overline{\mathrm{T}}, \langle\mathcal{B}_0\rangle\Delta_0) \qquad \langle\mathcal{B}_0\rangle\Delta_0 \vdash_{2} \langle\mathcal{A}\rangle\Gamma \to (\langle\overline{\mathcal{A}}\rangle\overline{\Gamma}, \langle\mathcal{B}_1\rangle\Delta_1) \\ \langle\overline{\mathcal{A}}_0\rangle\overline{\Gamma}_0 \triangleq \langle\overline{\mathcal{A}}\rangle\overline{\Gamma} \oplus x \mapsto (\alpha, \tau, \{(\overline{\mathrm{T}}, \sigma)\})\end{array}}{\langle\mathcal{B}\rangle\Delta \vdash_{2} \langle\{x\} \uplus \mathcal{A}\rangle\Gamma \to (\langle\overline{\mathcal{A}}_0\rangle\overline{\Gamma}_0, \langle\mathcal{B}_1\rangle\Delta_1)}$$

$$\langle\mathcal{B}\rangle\Delta \vdash_{2} \langle\varnothing\rangle\Gamma \to (\langle\varnothing\rangle\Gamma, \langle\mathcal{B}\rangle\Delta)$$

$$\frac{\begin{array}{c}y \notin \mathcal{B} \qquad \langle\mathcal{B}_0\rangle\Delta_0 \triangleq \langle\mathcal{B}\rangle\Delta \oplus y \mapsto ([\,], \mathsf{TRef}\,\text{"INTEGER"}, \{\}, v) \\ \langle\mathcal{B}_0\rangle\Delta_0 \vdash_{29} \mathsf{INTEGER}\,\mathcal{C} \to (\mathsf{INTEGER}\,\overline{\mathcal{C}}, \langle\mathcal{B}_1\rangle\Delta_1) \\ \overline{\mathrm{T}} \triangleq \mathsf{INTEGER}\,((c, \mathsf{VRef}\,y) :: \overline{\mathcal{C}})\end{array}}{\langle\mathcal{B}\rangle\Delta \vdash_{29} \mathsf{INTEGER}\,((c, v) :: \mathcal{C}) \to (\overline{\mathrm{T}}, \langle\mathcal{B}_1\rangle\Delta_1)}$$

$$\frac{\begin{array}{c} y \notin \mathcal{B} \qquad \langle \mathcal{B}_0 \rangle \Delta_0 \triangleq \langle \mathcal{B} \rangle \Delta \oplus y \mapsto ([\,], \mathsf{TRef}\,\text{"INTEGER"}, \{\}, v) \\ \langle \mathcal{B}_0 \rangle \Delta_0 \vdash_{29} \mathsf{ENUMERATED}\ \mathcal{C} \to (\mathsf{ENUMERATED}\ \overline{\mathcal{C}}, \langle \mathcal{B}_1 \rangle \Delta_1) \\ \overline{\mathrm{T}} \triangleq \mathsf{ENUMERATED}\ ((c, \mathsf{VRef}\, y) :: \overline{\mathcal{C}}) \end{array}}{\langle \mathcal{B} \rangle \Delta \vdash_{29} \mathsf{ENUMERATED}\ ((c, v) :: \mathcal{C}) \to (\overline{\mathrm{T}}, \langle \mathcal{B}_1 \rangle \Delta_1)}$$

$$\frac{\begin{array}{c} y \notin \mathcal{B} \qquad \langle \mathcal{B}_0 \rangle \Delta_0 \triangleq \langle \mathcal{B} \rangle \Delta \oplus y \mapsto ([\,], \mathsf{TRef}\,\text{"INTEGER"}, \{\}, v) \\ \langle \mathcal{B}_0 \rangle \Delta_0 \vdash_{29} \mathsf{BIT\ STRING}\ \mathcal{C} \to (\mathsf{BIT\ STRING}\ \overline{\mathcal{C}}, \langle \mathcal{B}_1 \rangle \Delta_1) \\ \overline{\mathrm{T}} \triangleq \mathsf{BIT\ STRING}\ ((c, \mathsf{VRef}\, y) :: \overline{\mathcal{C}}) \end{array}}{\langle \mathcal{B} \rangle \Delta \vdash_{29} \mathsf{BIT\ STRING}\ ((c, v) :: \mathcal{C}) \to (\overline{\mathrm{T}}, \langle \mathcal{B}_1 \rangle \Delta_1)}$$

$$\frac{\mathrm{T} \not\lhd \mathsf{INTEGER}\,(\_ :: \_) \mid \mathsf{BIT\ STRING}\,(\_ :: \_) \mid \mathsf{ENUMERATED}\,(\_ :: \_)}{\langle \mathcal{B} \rangle \Delta \vdash_{29} \mathrm{T} \to (\mathrm{T}, \langle \mathcal{B} \rangle \Delta)}$$

6.5.3 Découplage valeurs/types

$$\frac{\langle \mathcal{B} \rangle \Delta \vdash_{23} \langle \mathcal{A} \rangle \Gamma \to (\langle \mathcal{A}_0 \rangle \Gamma_0, \langle \mathcal{B}_0 \rangle \Delta_0) \qquad \langle \mathcal{B}_0 \rangle \Delta_0 \vdash_{2} \langle \mathcal{A}_0 \rangle \Gamma_0 \to r}{\langle \mathcal{B} \rangle \Delta \vdash_{48} \langle \mathcal{A} \rangle \Gamma \to r}$$

6.6 Découplage valeurs/contraintes

$$\frac{\langle \mathcal{A} \rangle \Gamma, \langle \mathcal{B} \rangle \Delta \vdash_{11} \langle \mathcal{A} \rangle \Gamma \to r}{\langle \mathcal{B} \rangle \Delta \vdash_{53} \langle \mathcal{A} \rangle \Gamma \to r}$$

$$\frac{\begin{array}{c} \Gamma(x) \lhd (\alpha, \tau, \{(\mathrm{T}, \sigma)\}) \qquad \langle \mathcal{A} \rangle \Gamma, \langle \mathcal{B} \rangle \Delta, x \vdash_{22} \sigma \to (\overline{\sigma}, \langle \mathcal{B}_0 \rangle \Delta_0) \\ \langle \mathcal{A} \rangle \Gamma, \langle \mathcal{B}_0 \rangle \Delta_0 \vdash_{11} \langle \mathcal{A}' \rangle \Gamma \to (\langle \overline{\mathcal{A}}' \rangle \overline{\Gamma}, \langle \mathcal{B}_1 \rangle \Delta_1) \\ \langle \overline{\mathcal{A}}_0 \rangle \overline{\Gamma}_0 \triangleq \langle \overline{\mathcal{A}}' \rangle \overline{\Gamma} \oplus x \mapsto (\alpha, \tau, \{(\mathrm{T}, \overline{\sigma})\}) \end{array}}{\langle \mathcal{A} \rangle \Gamma, \langle \mathcal{B} \rangle \Delta \vdash_{11} \langle \{x\} \uplus \mathcal{A}' \rangle \Gamma \to (\langle \overline{\mathcal{A}}_0 \rangle \overline{\Gamma}_0, \langle \mathcal{B}_1 \rangle \Delta_1)}$$

$$\langle \mathcal{A} \rangle \Gamma, \langle \mathcal{B} \rangle \Delta \vdash_{11} \langle \varnothing \rangle \Gamma \to (\langle \varnothing \rangle \Gamma, \langle \mathcal{B} \rangle \Delta)$$

$$\frac{\begin{array}{c} \langle \mathcal{A} \rangle \Gamma, \langle \mathcal{B} \rangle \Delta, x \vdash_{21} \nu \to (\overline{\nu}, \langle \mathcal{B}_0 \rangle \Delta_0) \\ \langle \mathcal{A} \rangle \Gamma, \langle \mathcal{B}_0 \rangle \Delta_0, x \vdash_{22} \sigma \to (\overline{\sigma}, \langle \mathcal{B}_1 \rangle \Delta_1) \end{array}}{\langle \mathcal{A} \rangle \Gamma, \langle \mathcal{B} \rangle \Delta, x \vdash_{22} \{\nu\} \uplus \sigma \to (\{\overline{\nu}\} \cup \overline{\sigma}, \langle \mathcal{B}_1 \rangle \Delta_1)}$$

$$\langle \mathcal{A} \rangle \Gamma, \langle \mathcal{B} \rangle \Delta, x \vdash_{22} \{\} \to (\{\}, \langle \mathcal{B} \rangle \Delta)$$

$$\frac{\langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, x \vdash_{22} \sigma' \to (\overline{\sigma}', \langle\mathcal{B}_0\rangle\Delta_0) \quad \langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}_0\rangle\Delta_0, x \vdash_{21} \mathsf{Inter}\,\Sigma' \to (\mathsf{Inter}\,\overline{\Sigma}', \langle\mathcal{B}_1\rangle\Delta_1) \quad \overline{\nu} \triangleq \mathsf{Inter}\,(\overline{\sigma}' :: \overline{\Sigma}')}{\langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, x \vdash_{21} \mathsf{Inter}\,(\sigma' :: \Sigma') \to (\overline{\nu}, \langle\mathcal{B}_1\rangle\Delta_1)}$$

$$\langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, x \vdash_{21} \mathsf{Inter}\,[\,] \to (\mathsf{Inter}\,[\,], \langle\mathcal{B}\rangle\Delta)$$

$$\frac{y \notin \mathcal{B} \quad \overline{\nu} \triangleq \mathsf{Value}\,(\mathsf{VRef}\,y) \quad \langle\mathcal{B}_0\rangle\Delta_0 \triangleq \langle\mathcal{B}\rangle\Delta \oplus y \mapsto ([\,], \mathsf{TRef}\,x, \{\}, v)}{\langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, x \vdash_{21} \mathsf{Value}\,(v) \to (\overline{\nu}, \langle\mathcal{B}_0\rangle\Delta_0)}$$

$$\frac{\nu \lhd (\mathsf{Val}\,(\mathsf{VRef}\ \ \_))\ \_..\_\ (\mathsf{Val}\,(\mathsf{VRef}\ \ \_))}{\langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, x \vdash_{21} \nu \to (\nu, \langle\mathcal{B}\rangle\Delta)}$$

$$\frac{v_0 \not\lhd \mathsf{VRef}\ \ \_ \quad y_0 \notin \mathcal{B} \quad \nu \triangleq (\mathsf{Val}\,(\mathsf{VRef}\,y_0))\,b_0..b_1e_1 \quad \delta \triangleq y_0 \mapsto ([\,], \mathsf{TRef}\,x, \{\}, v_0) \quad \langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta \oplus \delta, x \vdash_{21} \nu \to r}{\langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, x \vdash_{21} (\mathsf{Val}\,v_0)\,b_0..b_1e_1 \to r}$$

$$\frac{v_1 \not\lhd \mathsf{VRef}\ \ \_ \quad y_1 \notin \mathcal{B} \quad \nu \triangleq e_0b_0..b_1\,(\mathsf{Val}\,(\mathsf{VRef}\,y_1)) \quad \delta \triangleq y_1 \mapsto ([\,], \mathsf{TRef}\,x, \{\}, v_1) \quad \langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta \oplus \delta, x \vdash_{21} \nu \to r}{\langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, x \vdash_{21} e_0b_0..b_1\,(\mathsf{Val}\,v_1) \to r}$$

$$\frac{\langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, \text{"INTEGER"} \vdash_{22} \sigma \to (\overline{\sigma}, \langle\mathcal{B}_0\rangle\Delta_0)}{\langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, x \vdash_{21} \mathsf{SIZE}\,\sigma \to (\mathsf{SIZE}\,\overline{\sigma}, \langle\mathcal{B}_0\rangle\Delta_0)}$$

$$\frac{\langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, x \vdash_{22} \sigma \to (\overline{\sigma}, \langle\mathcal{B}_0\rangle\Delta_0)}{\langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, x \vdash_{21} \mathsf{FROM}\ \sigma \to (\mathsf{FROM}\ \overline{\sigma}, \langle\mathcal{B}_0\rangle\Delta_0)}$$

$$\frac{x' \in \mathcal{A} \quad \Gamma(x') \lhd (\alpha', \tau', \{(\mathrm{T}', \sigma')\}) \quad \mathrm{T} \lhd (\mathsf{SET\ OF} \mid \mathsf{SEQUENCE\ OF})\ [\,]\ (\mathsf{TRef}\,x'')\ \{\} \quad \langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, x'' \vdash_{22} \sigma \to (\overline{\sigma}, \langle\mathcal{B}_0\rangle\Delta_0) \quad \overline{\nu} \triangleq \mathsf{WITH\ COMPONENT}\ \overline{\sigma}}{\langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, x' \vdash_{21} \mathsf{WITH\ COMPONENT}\ \sigma \to (\overline{\nu}, \langle\mathcal{B}_0\rangle\Delta_0)}$$

$$\frac{\begin{array}{c} x' \in \mathcal{A} \qquad \Gamma(x') \lhd (\alpha', \tau', \{(\mathrm{T}', \sigma')\}) \\ \mathrm{T}' \lhd (\mathsf{SET} \mid \mathsf{SEQUENCE})\ [\mathsf{Field}\,(l'_i, [\,], \mathsf{TRef}\,x'_i, \{\}, s'_i)]_{1 \leqslant i \leqslant n} \\ k' \lhd (l, \sigma, \hat{\pi}) \\ \exists i \in [1..n].(l = l'_i \wedge \langle \mathcal{A} \rangle \Gamma, \langle \mathcal{B} \rangle \Delta, x'_i \vdash_{\overline{2}2} \sigma \to (\overline{\sigma}, \langle \mathcal{B}_0 \rangle \Delta_0)) \\ \nu' \triangleq \mathsf{WITH\ COMPONENTS}\ (m, \mathcal{K}') \\ \langle \mathcal{A} \rangle \Gamma, \langle \mathcal{B}_0 \rangle \Delta_0, x' \vdash_{\overline{2}1} \nu' \to (\overline{\nu}', \langle \mathcal{B}_1 \rangle \Delta_1) \\ \overline{\nu}' \lhd \mathsf{WITH\ COMPONENTS}\ (\_, \overline{\mathcal{K}}') \\ \overline{\nu} \triangleq \mathsf{WITH\ COMPONENTS}\ (m, (l, \overline{\sigma}, \hat{\pi}) :: \overline{\mathcal{K}}') \end{array}}{\langle \mathcal{A} \rangle \Gamma, \langle \mathcal{B} \rangle \Delta, x' \vdash_{\overline{2}1} \mathsf{WITH\ COMPONENTS}\ (m, k' :: \mathcal{K}') \to (\overline{\nu}, \langle \mathcal{B}_1 \rangle \Delta_1)}$$

$$\frac{\begin{array}{c} x' \in \mathcal{A} \qquad \Gamma(x') \lhd (\alpha', \tau', \{(\mathrm{T}', \sigma')\}) \\ \mathrm{T}' \lhd \mathsf{CHOICE}\,[(l'_i, [\,], \mathsf{TRef}\,x'_i, \{\}, s'_i)]_{1 \leqslant i \leqslant n} \qquad k' \lhd (l, \sigma, \hat{\pi}) \\ \exists i \in [1..n].(l = l'_i \wedge \langle \mathcal{A} \rangle \Gamma, \langle \mathcal{B} \rangle \Delta, x'_i \vdash_{\overline{2}2} \sigma \to (\overline{\sigma}, \langle \mathcal{B}_0 \rangle \Delta_0)) \\ \nu' \triangleq \mathsf{WITH\ COMPONENTS}\ (m, \mathcal{K}') \\ \langle \mathcal{A} \rangle \Gamma, \langle \mathcal{B}_0 \rangle \Delta_0, x' \vdash_{\overline{2}1} \nu' \to (\overline{\nu}', \langle \mathcal{B}_1 \rangle \Delta_1) \\ \overline{\nu}' \lhd \mathsf{WITH\ COMPONENTS}\ (\_, \overline{\mathcal{K}}') \\ \overline{\nu} \triangleq \mathsf{WITH\ COMPONENTS}\ (m, (l, \overline{\sigma}, \hat{\pi}) :: \overline{\mathcal{K}}') \end{array}}{\langle \mathcal{A} \rangle \Gamma, \langle \mathcal{B} \rangle \Delta, x' \vdash_{\overline{2}1} \mathsf{WITH\ COMPONENTS}\ (m, k' :: \mathcal{K}') \to (\overline{\nu}, \langle \mathcal{B}_1 \rangle \Delta_1)}$$

$$\frac{\nu \lhd \mathsf{WITH\ COMPONENTS}\ (m, [\,])}{\langle \mathcal{A} \rangle \Gamma, \langle \mathcal{B} \rangle \Delta, x' \vdash_{\overline{2}1} \nu \to (\nu, \langle \mathcal{B} \rangle \Delta)}$$

6.7 Normalisation des valeurs

6.7.1 Désambigüation du zéro

$$\frac{\begin{array}{c} \Delta(y) \lhd (\tau, \mathrm{T}, \sigma, v) \\ \langle \mathcal{A} \rangle \Gamma, [\,], \text{""} \vdash_{\overline{8}3} v : \mathrm{T} \to \overline{v} \qquad \langle \mathcal{A} \rangle \Gamma \vdash_{\overline{8}4} \langle \mathcal{B} \rangle \Delta \to \langle \overline{\mathcal{B}} \rangle \overline{\Delta} \end{array}}{\langle \mathcal{A} \rangle \Gamma \vdash_{\overline{8}4} \langle \{y\} \uplus \mathcal{B} \rangle \Delta \to \langle \overline{\mathcal{B}} \rangle \overline{\Delta} \oplus y \mapsto (\tau, \mathrm{T}, \sigma, \overline{v})}$$

$$\langle \mathcal{A} \rangle \Gamma \vdash_{\overline{8}4} \langle \varnothing \rangle \Delta \to \langle \varnothing \rangle \Delta$$

$$\frac{x' \in \mathcal{A} \qquad \Gamma(x') \lhd (\alpha', \tau', \{(\mathrm{T}', \sigma')\}) \qquad \langle \mathcal{A} \rangle \Gamma, \mathrm{H}, x' \vdash_{\overline{8}3} v : \mathrm{T}' \to \overline{v}}{\langle \mathcal{A} \rangle \Gamma, \mathrm{H}, x \vdash_{\overline{8}3} v : (\mathsf{TRef}\,x') \to \overline{v}}$$

$$\langle \mathcal{A} \rangle \Gamma, \mathrm{H}, \text{"REAL"} \vdash_{\overline{8}3} \mathsf{Zero} : \mathrm{T} \to 0.0$$

$$\langle \mathcal{A} \rangle \Gamma, \mathrm{H}, x \vdash_{\overline{8}3} \mathsf{Zero} : \mathsf{INTEGER}\ \ \_ \to \mathsf{Int}\,(0)$$

$$\frac{\mathrm{T} \lhd (\mathsf{SEQUENCE\ OF} \mid \mathsf{SET\ OF})\,\tau'\,\mathrm{T}'\,\sigma' \qquad v \notin \mathrm{H} \qquad \langle\mathcal{A}\rangle\Gamma, v :: \mathrm{H}, \text{""} \vdash_{\overline{\mathrm{83}}} v' : \mathrm{T}' \to \overline{v}' \qquad \langle\mathcal{A}\rangle\Gamma, \mathrm{H}, x \vdash_{\overline{\mathrm{83}}} \{V\} : \mathrm{T} \to \{\overline{V}\}}{\langle\mathcal{A}\rangle\Gamma, \mathrm{H}, x \vdash_{\overline{\mathrm{83}}} (\{(\mathsf{None}, v') :: V\}\ \mathbf{as}\ v) : \mathrm{T} \to \{(\mathsf{None}, \overline{v}') :: \overline{V}\}}$$

$$\frac{\mathrm{T} \lhd \mathsf{CHOICE}\,((l', \tau', \mathrm{T}', \sigma', s') :: \mathcal{F}') \qquad x' = l' \qquad v \notin \mathrm{H} \qquad \langle\mathcal{A}\rangle\Gamma, v :: \mathrm{H}, \text{""} \vdash_{\overline{\mathrm{83}}} v' : \mathrm{T}' \to \overline{v}'}{\langle\mathcal{A}\rangle\Gamma, \mathrm{H}, x \vdash_{\overline{\mathrm{83}}} ((x' : v')\ \mathbf{as}\ v) : \mathrm{T} \to x' : \overline{v}'}$$

$$\frac{\mathrm{T} \lhd \mathsf{CHOICE}\,((l', \tau', \mathrm{T}', \sigma', s') :: \mathcal{F}') \qquad x' \neq l' \qquad \langle\mathcal{A}\rangle\Gamma, \mathrm{H}, x \vdash_{\overline{1}} (x' : v') : \mathsf{CHOICE}\,\mathcal{F}' \to \overline{v}}{\langle\mathcal{A}\rangle\Gamma, \mathrm{H}, x \vdash_{\overline{\mathrm{83}}} (x' : v') : \mathrm{T} \to \overline{v}}$$

$$\frac{\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \qquad V \lhd [(\mathsf{Some}\,l', v')] \sqcup V' \qquad v \notin \mathrm{H} \qquad \langle\mathcal{A}\rangle\Gamma, \mathrm{H}, x \vdash_{\overline{1}} \{V'\} : \mathsf{SET}\,\Phi' \to \{\overline{V}'\} \qquad \langle\mathcal{A}\rangle\Gamma, v :: \mathrm{H}, \text{""} \vdash_{\overline{\mathrm{83}}} v' : \mathrm{T}' \to \overline{v}' \qquad \overline{V} \triangleq [(\mathsf{Some}\,l', \overline{v}')] \sqcup \overline{V}'}{\langle\mathcal{A}\rangle\Gamma, \mathrm{H}, x \vdash_{\overline{\mathrm{83}}} (\{V\}\ \mathbf{as}\ v) : \mathsf{SET}\,(\varphi' :: \Phi') \to \{\overline{V}\}}$$

$$\frac{\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \qquad \forall \hat{v} \in V.(\hat{v} \lhd (\mathsf{Some}\,x', v') \wedge x' \neq l') \qquad \langle\mathcal{A}\rangle\Gamma, \mathrm{H}, x \vdash_{\overline{\mathrm{83}}} \{V\} : \mathsf{SET}\,\Phi' \to \overline{v}}{\langle\mathcal{A}\rangle\Gamma, \mathrm{H}, x \vdash_{\overline{\mathrm{83}}} \{V\} : \mathsf{SET}\,(\varphi' :: \Phi') \to \overline{v}}$$

$$\frac{\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \qquad V \lhd (\mathsf{Some}\,x', v') :: V' \qquad x' = l' \qquad v \notin \mathrm{H} \qquad \langle\mathcal{A}\rangle\Gamma, \mathrm{H}, x \vdash_{\overline{1}} \{V'\} : \mathsf{SEQUENCE}\,\Phi' \to \{\overline{V}'\} \qquad \langle\mathcal{A}\rangle\Gamma, v :: \mathrm{H}, \text{""} \vdash_{\overline{\mathrm{83}}} v' : \mathrm{T}' \to \overline{v}' \qquad \overline{V} \triangleq (\mathsf{Some}\,x', \overline{v}') :: \overline{V}'}{\langle\mathcal{A}\rangle\Gamma, \mathrm{H}, x \vdash_{\overline{\mathrm{83}}} (\{V\}\ \mathbf{as}\ v) : \mathsf{SEQUENCE}\,(\varphi' :: \Phi') \to \{\overline{V}\}}$$

$$\frac{\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \qquad V \lhd (\mathsf{Some}\,x', v') :: V' \qquad x' \neq l' \qquad \langle\mathcal{A}\rangle\Gamma, \mathrm{H}, x \vdash_{\overline{\mathrm{83}}} \{V\} : \mathsf{SEQUENCE}\,\Phi' \to \overline{v}}{\Gamma, \mathrm{H}, x \vdash_{\overline{\mathrm{83}}} \{V\} : \mathsf{SEQUENCE}\,(\varphi' :: \Phi') \to \overline{v}}$$

$$\frac{\mathbf{when}\ \textit{no other rule match}}{\langle\mathcal{A}\rangle\Gamma, \mathrm{H}, x \vdash_{\overline{\mathrm{83}}} v : \mathrm{T} \to v}$$

6.7.2 Forme normale

$$\frac{\Delta'(y) \lhd ([\,], \mathsf{TRef}\, x, \{\}, v) \qquad \langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, [\,] \vdash_{\overline{85}} v : \mathsf{TRef}\,(x) \to \overline{v} \qquad \langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta \vdash_{\overline{17}} \langle\mathcal{B}'\rangle\Delta' \to \langle\overline{\mathcal{B}'}\rangle\overline{\Delta'}}{\langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta \vdash_{\overline{17}} \langle\{y\} \uplus \mathcal{B}'\rangle\Delta' \to \langle\overline{\mathcal{B}'}\rangle\overline{\Delta'} \oplus y \mapsto ([\,], \mathsf{TRef}\, x, \{\}, \overline{v})}$$

$$\langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta \vdash_{\overline{17}} \langle\varnothing\rangle\Delta \to \langle\varnothing\rangle\Delta$$

$$\frac{\begin{array}{c} x \in \mathcal{A} \qquad \Gamma(x) \lhd (\alpha, \tau, \{(\mathrm{T}, \sigma)\}) \qquad \mathrm{T} \lhd \mathsf{ENUMERATED}\ [(c_j, v_j)]_{1 \leqslant j \leqslant p} \\ \forall j \in [1..p].y' \neq c_j \qquad y' \in \mathcal{B} \qquad \Delta(y') \lhd ([\,], \mathsf{TRef}\, x', \{\}, v') \\ x' \in \mathcal{A} \qquad \Gamma(x') \lhd (\alpha', \tau', \{(\mathrm{T}', \sigma')\}) \qquad x = x' \vee \alpha \sqcap \alpha' \neq \{\} \\ v \notin \mathrm{H} \qquad \langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, v :: \mathrm{H} \vdash_{\overline{85}} v' : \mathsf{TRef}\,(x') \to \overline{v}' \end{array}}{\langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, \mathrm{H} \vdash_{\overline{85}} (\mathsf{Id}\,(y')\ \mathbf{as}\ v) : \mathsf{TRef}\,(x) \to \overline{v}'}$$

$$\frac{x' \in \mathcal{A} \qquad \Gamma(x') \lhd (\alpha', \tau', \{(\mathrm{T}', \sigma')\}) \qquad \langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, \mathrm{H} \vdash_{\overline{85}} v : \mathrm{T}' \to \overline{v}}{\langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, \mathrm{H} \vdash_{\overline{85}} v : \mathsf{TRef}\,(x') \to \overline{v}}$$

$$\frac{\begin{array}{c} \mathrm{T} \lhd \mathsf{INTEGER}\ [(c_j, v_j)]_{1 \leqslant j \leqslant p} \\ \forall j \in [1..p].y' \neq c_j \qquad y' \in \mathcal{B} \qquad \Delta(y') \lhd ([\,], \mathsf{TRef}\, x', \{\}, v') \\ v \notin \mathrm{H} \qquad \langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, v :: \mathrm{H} \vdash_{\overline{85}} v' : \mathsf{TRef}\,(x') \to \overline{v}' \end{array}}{\langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, \mathrm{H} \vdash_{\overline{85}} (\mathsf{Id}\,(y')\ \mathbf{as}\ v) : \mathrm{T} \to \overline{v}'}$$

$$\frac{\exists j \in [1..p].(y' = c_j \wedge \langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, \mathrm{H} \vdash_{\overline{85}} v_j : \mathsf{INTEGER}\ [\,] \to \overline{v})}{\langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, \mathrm{H} \vdash_{\overline{85}} \mathsf{Id}\,(y') : \mathsf{INTEGER}\,[(c_j, v_j)]_{1 \leqslant j \leqslant p} \to \overline{v}}$$

$$\frac{\exists j \in [1..p].y' = c_j}{\langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, \mathrm{H} \vdash_{\overline{85}} \mathsf{Id}\,(y') : \mathsf{ENUMERATED}\,[(c_j, v_j)]_{1 \leqslant j \leqslant p} \to \mathsf{Enum}\,(y')}$$

$$\frac{\begin{array}{c} y' \in \mathcal{B} \qquad \Delta(y') \lhd ([\,], \mathsf{TRef}\, x', \{\}, v') \\ v \notin \mathrm{H} \qquad \langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, v :: \mathrm{H} \vdash_{\overline{85}} v' : \mathrm{T}' \to \overline{v}' \end{array}}{\langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, \mathrm{H} \vdash_{\overline{85}} (\mathsf{Id}\,(y')\ \mathbf{as}\ v) : \mathrm{T} \to \overline{v}'}$$

$$\frac{\begin{array}{c} \mathrm{T} \lhd \mathsf{BIT\ STRING}\,[(c_j, v_j)]_{1 \leqslant j \leqslant p} \qquad \forall i \in [1..n].\exists j \in [1..p].y_i = c_j \\ \forall j \in [1..p].\langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, \mathrm{H} \vdash_{\overline{85}} v_j : \mathsf{INTEGER}\ [\,] \to \mathsf{Int}\,(n_j) \\ bs \triangleq \mathrm{positions\_to\_bits}\ [y_i]_{1 \leqslant i \leqslant n}\,[(c_j, n_j)]_{1 \leqslant j \leqslant p} \end{array}}{\langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, \mathrm{H} \vdash_{\overline{85}} \mathbf{\{}[(\mathsf{None}, \mathsf{Id}\, y_i)]_{1 \leqslant i \leqslant n}\mathbf{\}} : \mathrm{T} \to \mathsf{BinStr}\,(bs)}$$

$$\frac{\mathrm{T} \lhd \mathsf{BIT\ STRING}\ \_}{\langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, \mathrm{H} \vdash_{\overline{85}} \mathbf{\{}[\,]\mathbf{\}} : \mathrm{T} \to \mathsf{BinStr}\,(\text{""})}$$

$$\frac{\mathrm{T} \lhd \mathsf{BIT\ STRING}\ \ \_ \mid \mathsf{OCTET\ STRING} \qquad bs \triangleq \mathrm{hexa\_to\_bin}\,(hs)}{\langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, \mathrm{H} \vdash_{\overline{85}} \mathsf{HexaStr}\,(hs) : \mathrm{T} \to \mathsf{BinStr}\,(bs)}$$

$$\frac{\begin{array}{c}\mathrm{T} \lhd \mathsf{SEQUENCE\ OF}\,\tau'\,\mathrm{T}'\,\sigma' \qquad v \notin \mathrm{H} \\ \langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, v :: \mathrm{H} \vdash_{\overline{85}} v' : \mathrm{T}' \to \overline{v}' \qquad \langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, \mathrm{H} \vdash_{\overline{85}} \{V\} : \mathrm{T} \to \{\overline{V}\}\end{array}}{\langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, \mathrm{H} \vdash_{\overline{85}} (\{(\mathsf{None}, v') :: V\}\ \mathbf{as}\ v) : \mathrm{T} \to \{(\mathsf{None}, \overline{v}') :: \overline{V}\}}$$

$$\frac{\begin{array}{c}\mathrm{T} \lhd \mathsf{SET\ OF}\,\tau'\,\mathrm{T}'\,\sigma' \\ v \notin \mathrm{H} \qquad \forall i \in [1..n].\langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, v :: \mathrm{H} \vdash_{\overline{85}} v'_i : \mathrm{T}'_i \to \overline{v}'_i \\ \overline{v} \triangleq \{[(\mathsf{None}, \overline{v}'_{\pi(i)})]_{1 \leqslant i \leqslant n}\} \qquad \pi\ \textit{est une permutation sur}\ [1..n]\end{array}}{\langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, \mathrm{H} \vdash_{\overline{85}} (\{[(\mathsf{None}, v'_i)]_{1 \leqslant i \leqslant n}\}\ \mathbf{as}\ v) : \mathrm{T} \to \overline{v}}$$

$$\frac{\begin{array}{c}\mathrm{T} \lhd \mathsf{CHOICE}\,((l', \tau', \mathrm{T}', \sigma', s') :: \mathcal{F}') \\ x' = l' \qquad v \notin \mathrm{H} \qquad \langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, v :: \mathrm{H} \vdash_{\overline{85}} v' : \mathrm{T}' \to \overline{v}'\end{array}}{\langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, \mathrm{H} \vdash_{\overline{85}} ((x' : v')\ \mathbf{as}\ v) : \mathrm{T} \to x' : \overline{v}'}$$

$$\frac{\begin{array}{c}\mathrm{T} \lhd \mathsf{CHOICE}\,((l', \tau', \mathrm{T}', \sigma', s') :: \mathcal{F}') \\ x' \neq l' \qquad \langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, \mathrm{H} \vdash_{\overline{85}} (x' : v') : \mathsf{CHOICE}\,\mathcal{F}' \to \overline{v}\end{array}}{\langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, \mathrm{H} \vdash_{\overline{85}} (x' : v') : \mathrm{T} \to \overline{v}}$$

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \\ V \lhd [(\mathsf{Some}\,l', v')] \sqcup V' \qquad v \notin \mathrm{H} \qquad \langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, v :: \mathrm{H} \vdash_{\overline{85}} v' : \mathrm{T}' \to \overline{v}' \\ \langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, \mathrm{H} \vdash_{\overline{85}} \{V'\} : \mathsf{SET}\,\Phi' \to \{\overline{V}'\} \qquad \overline{V} \triangleq [(\mathsf{Some}\,l', \overline{v}')] \sqcup \overline{V}'\end{array}}{\langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, \mathrm{H} \vdash_{\overline{85}} \{V\} : \mathsf{SET}\,(\varphi' :: \Phi') \to \{\overline{V}\}}$$

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \\ \forall \hat{v} \in V.(\hat{v} \lhd (\mathsf{Some}\,x', \_) \wedge x' \neq l') \qquad s' \lhd \mathsf{Some}\,(\mathsf{DEFAULT}\ v') \\ v \notin \mathrm{H} \qquad \langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, v :: \mathrm{H} \vdash_{\overline{85}} v' : \mathrm{T}' \to \overline{v}' \\ \langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, \mathrm{H} \vdash_{\overline{85}} v : \mathsf{SET}\,\Phi' \to \{\overline{V}'\} \qquad \overline{V} \triangleq [(\mathsf{Some}\,l', \overline{v}')] \sqcup \overline{V}'\end{array}}{\langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, \mathrm{H} \vdash_{\overline{85}} (\{V\}\ \mathbf{as}\ v) : \mathsf{SET}\,(\varphi' :: \Phi') \to \{\overline{V}\}}$$

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \qquad \forall \hat{v} \in V.(\hat{v} \lhd (\mathsf{Some}\,x', \_) \wedge x' \neq l') \\ s' \not\lhd \mathsf{Some}\,(\mathsf{DEFAULT}\ \_) \qquad \langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, \mathrm{H} \vdash_{\overline{85}} \{V\} : \mathsf{SET}\,\Phi' \to \overline{v}\end{array}}{\langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, \mathrm{H} \vdash_{\overline{85}} \{V\} : \mathsf{SET}\,(\varphi' :: \Phi') \to \overline{v}}$$

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \qquad V \lhd (\mathsf{Some}\,x', v') :: V' \\ x' = l' \qquad v \notin \mathrm{H} \qquad \langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, v :: \mathrm{H} \vdash_{\overline{85}} v' : \mathrm{T}' \to \overline{v}' \\ \langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, \mathrm{H} \vdash_{\overline{85}} \{V'\} : \mathsf{SEQUENCE}\,\Phi' \to \{\overline{V}'\} \\ \overline{V} \triangleq (\mathsf{Some}\,x', \overline{v}') :: \overline{V}'\end{array}}{\langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, \mathrm{H} \vdash_{\overline{85}} (\{V\}\,\mathbf{as}\,v) : \mathsf{SEQUENCE}\,(\varphi' :: \Phi') \to \{\overline{V}\}}$$

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \\ V \lhd (\mathsf{Some}\,x', \_) :: V' \qquad x' \neq l' \qquad s' \lhd \mathsf{Some}\,(\mathsf{DEFAULT}\,v') \\ v \notin \mathrm{H} \qquad \langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, v :: \mathrm{H} \vdash_{\overline{85}} v' : \mathrm{T}' \to \overline{v}' \\ \langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, \mathrm{H} \vdash_{\overline{85}} v : \mathsf{SEQUENCE}\,\Phi' \to \{\overline{V}'\} \qquad \overline{V} \triangleq (\mathsf{Some}\,x', \overline{v}') :: \overline{V}'\end{array}}{\langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, \mathrm{H} \vdash_{\overline{85}} (\{V\}\,\mathbf{as}\,v) : \mathsf{SEQUENCE}\,(\varphi' :: \Phi') \to \{\overline{V}\}}$$

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \\ V \lhd (\mathsf{Some}\,x', \_) :: V' \qquad x' \neq l' \qquad s' \not\lhd \mathsf{Some}\,(\mathsf{DEFAULT}\,\_) \\ \langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, \mathrm{H} \vdash_{\overline{85}} \{V\} : \mathsf{SEQUENCE}\,\Phi' \to \overline{v}\end{array}}{\langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, \mathrm{H} \vdash_{\overline{85}} \{V\} : \mathsf{SEQUENCE}\,(\varphi' :: \Phi') \to \overline{v}}$$

$$\frac{\mathbf{when}\ \textit{no other rule match}}{\langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta, \mathrm{H} \vdash_{\overline{85}} v : \mathrm{T} \to v}$$

6.7.3 Normalisation des valeurs

$$\frac{\langle\mathcal{A}\rangle \vdash_{\overline{84}} \langle\mathcal{B}\rangle\Delta \to \langle\mathcal{B}_0\rangle\Delta_0 \qquad \langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}_0\rangle\Delta_0 \vdash_{\overline{17}} \langle\mathcal{B}_0\rangle\Delta_0 \to \langle\overline{\mathcal{B}}\rangle\overline{\Delta}}{\langle\mathcal{A}\rangle\Gamma \vdash_{\overline{18}} \langle\mathcal{B}\rangle\Delta \to \langle\overline{\mathcal{B}}\rangle\overline{\Delta}}$$

$$\langle\mathcal{A}\rangle\Gamma \vdash_{\overline{18}} \langle\varnothing\rangle\Delta \to \langle\varnothing\rangle\Delta$$

6.8 Recouplage valeurs/types

Les valeurs par défaut sont ignorées.

$$\frac{\Gamma(x) \lhd (\alpha, \tau, \{(\mathrm{T}, \sigma)\}) \qquad \langle\mathcal{B}\rangle\Delta \vdash_{\overline{97}} \mathrm{T} \to \overline{\mathrm{T}} \qquad \langle\mathcal{B}\rangle\Delta \vdash_{\overline{96}} \langle\mathcal{A}\rangle\Gamma \to \langle\overline{\mathcal{A}}\rangle\overline{\Gamma}}{\langle\mathcal{B}\rangle\Delta \vdash_{\overline{96}} \langle\{x\} \uplus \mathcal{A}\rangle\Gamma \to \langle\overline{\mathcal{A}}\rangle\overline{\Gamma} \oplus x \mapsto (\alpha, \tau, \{(\overline{\mathrm{T}}, \sigma)\})}$$

$$\langle\mathcal{B}\rangle\Delta \vdash_{\overline{96}} \langle\varnothing\rangle\Gamma \to \langle\varnothing\rangle\Gamma$$

$$\frac{\begin{array}{c}y \in \mathcal{B} \qquad \Delta(y) \lhd ([\,], \mathsf{TRef}\ \_, \{\}, v) \\ \langle\mathcal{B}\rangle\Delta \vdash_{\overline{97}} \mathsf{INTEGER}\,\mathcal{C} \to \mathsf{INTEGER}\,\overline{\mathcal{C}} \qquad \overline{\mathrm{T}} \triangleq \mathsf{INTEGER}\ ((c, v) :: \overline{\mathcal{C}})\end{array}}{\langle\mathcal{B}\rangle\Delta \vdash_{\overline{97}} \mathsf{INTEGER}\ ((c, \mathsf{VRef}\,y) :: \mathcal{C}) \to \overline{\mathrm{T}}}$$

$$\frac{\begin{array}{c} y \in \mathcal{B} \qquad \Delta(y) \lhd ([\,], \mathsf{TRef}\ \_, \{\}, v) \\ \langle \mathcal{B} \rangle \Delta \vdash_{97} \mathsf{ENUMERATED}\ \mathcal{C} \to \mathsf{ENUMERATED}\ \overline{\mathcal{C}} \\ \overline{\mathrm{T}} \triangleq \mathsf{ENUMERATED}\ ((c, v) :: \overline{\mathcal{C}}) \end{array}}{\langle \mathcal{B} \rangle \Delta \vdash_{97} \mathsf{ENUMERATED}\ ((c, \mathsf{VRef}\, y) :: \mathcal{C}) \to \overline{\mathrm{T}}}$$

$$\frac{\begin{array}{c} y \in \mathcal{B} \qquad \Delta(y) \lhd ([\,], \mathsf{TRef}\ \_, \{\}, v) \\ \langle \mathcal{B} \rangle \Delta \vdash_{97} \mathsf{BIT\ STRING}\ \mathcal{C} \to \mathsf{BIT\ STRING}\ \overline{\mathcal{C}} \\ \overline{\mathrm{T}} \triangleq \mathsf{BIT\ STRING}\ ((c, v) :: \overline{\mathcal{C}}) \end{array}}{\langle \mathcal{B} \rangle \Delta \vdash_{97} \mathsf{BIT\ STRING}\ ((c, \mathsf{VRef}\, y) :: \mathcal{C}) \to \overline{\mathrm{T}}}$$

$$\frac{\mathrm{T} \not\lhd \mathsf{INTEGER}\,(\_ :: \_) \mid \mathsf{BIT\ STRING}\,(\_ :: \_) \mid \mathsf{ENUMERATED}\,(\_ :: \_)}{\Delta \vdash_{97} \mathrm{T} \to \mathrm{T}}$$

6.9 Unicité et positivité de constantes

$$\frac{\Gamma(x) \lhd (\alpha, \tau, \{(\mathrm{T}, \sigma)\}) \qquad \vdash_{28} \mathrm{T} \qquad \vdash_{30} \langle \mathcal{A} \rangle \Gamma}{\vdash_{30} \langle \{x\} \uplus \mathcal{A} \rangle \Gamma} \qquad \vdash_{30} \langle \varnothing \rangle \Gamma$$

$$\frac{\begin{array}{c} \forall j \in [1..p].n_j \geqslant 0 \\ \forall (i,j) \in [1..p]^2.(c_i = c_j \Rightarrow i = j) \qquad \forall (i,j) \in [1..p]^2.(n_i = n_j \Rightarrow i = j) \end{array}}{\vdash_{28} (\mathsf{ENUMERATED} \mid \mathsf{BIT\ STRING})\,[(c_j, \mathsf{Int}\, n_j)]_{1 \leqslant j \leqslant p}}\ [1]$$

$$\frac{\forall (i,j) \in [1..p]^2.(c_i = c_j \Rightarrow i = j)}{\vdash_{28} \mathsf{INTEGER}\,[(c_j, v_j)]_{1 \leqslant j \leqslant p}}\ [2]$$

$$\frac{\mathrm{T} \not\lhd \mathsf{INTEGER}\,(\_ :: \_) \mid \mathsf{ENUMERATED}\ \_ \mid \mathsf{BIT\ STRING}\ \_}{\vdash_{28} \mathrm{T}}\ [3]$$

6.10 Normalisation des étiquetages

6.10.1 Globalisation des étiquetages

Première étape : réécriture des types et récolte des étiquetages locaux avec leur abréviation associée.

$$\frac{\begin{array}{c} \Gamma(x) \lhd (\alpha, \tau, \{(\mathrm{T}, \sigma)\}) \qquad \langle \mathcal{X} \rangle \Xi \vdash_{3} \mathrm{T} \to (\overline{\mathrm{T}}, \langle \mathcal{X}_0 \rangle \Xi_0) \\ \langle \mathcal{X}_0 \rangle \Xi_0 \vdash_{49} \langle \mathcal{A} \rangle \Gamma \to (\langle \overline{\mathcal{A}} \rangle \overline{\Gamma}, \langle \mathcal{X}_1 \rangle \Xi_1) \qquad \overline{\gamma} \triangleq x \mapsto (\alpha, \tau, \{(\overline{\mathrm{T}}, \sigma)\}) \end{array}}{\langle \mathcal{X} \rangle \Xi \vdash_{49} \langle \{x\} \uplus \mathcal{A} \rangle \Gamma \to (\langle \overline{\mathcal{A}} \rangle \overline{\Gamma} \oplus \overline{\gamma}, \langle \mathcal{X}_1 \rangle \Xi_1)}$$

$$\langle \mathcal{X} \rangle \Xi \vdash_{49} \langle \varnothing \rangle \to (\langle \varnothing \rangle, \langle \mathcal{X} \rangle \Xi)$$

$$\frac{\tau' \not\lhd [\,] \qquad \overline{\mathrm{T}} \triangleq \mathsf{SET\ OF}\ [\,]\ (\mathsf{TRef}\, x)\ \{\}}{\langle \mathcal{X} \rangle \Xi \vDash_3 \mathsf{SET\ OF}\, \tau'\ (\mathsf{TRef}\, x)\ \{\} \to (\overline{\mathrm{T}}, \langle \mathcal{X} \rangle \Xi \oplus x \mapsto \tau')}$$

$$\frac{\mathrm{T} \lhd \mathsf{SET}\, [\,]\ \mathrm{T}'\ \{\}}{\langle \mathcal{X} \rangle \Xi \vDash_3 \mathrm{T} \to (\mathrm{T}, \langle \mathcal{X} \rangle \Xi)}$$

$$\frac{\tau' \not\lhd [\,] \qquad \overline{\mathrm{T}} \triangleq \mathsf{SEQUENCE\ OF}\ [\,]\ (\mathsf{TRef}\, x)\ \{\}}{\langle \mathcal{X} \rangle \Xi \vDash_3 \mathsf{SEQUENCE\ OF}\, \tau'\ (\mathsf{TRef}\, x)\ \{\} \to (\overline{\mathrm{T}}, \langle \mathcal{X} \rangle \Xi \oplus x \mapsto \tau')}$$

$$\frac{\mathrm{T} \lhd \mathsf{SEQUENCE}\, [\,]\ \mathrm{T}'\ \{\}}{\langle \mathcal{X} \rangle \Xi \vDash_3 \mathrm{T} \to (\mathrm{T}, \langle \mathcal{X} \rangle \Xi)}$$

$$\frac{\begin{array}{c} f' \lhd (l', \tau', \mathsf{TRef}\, x, \{\}, s') \\ \tau' \not\lhd [\,] \qquad \langle \mathcal{X} \rangle \Xi \vDash_3 \mathsf{CHOICE}\, \mathcal{F}' \to (\mathsf{CHOICE}\, \overline{\mathcal{F}}', \langle \mathcal{X}_0 \rangle \Xi_0) \\ \overline{\mathrm{T}} \triangleq \mathsf{CHOICE}\, ((l', [\,], \mathsf{TRef}\, x, \{\}, s') :: \overline{\mathcal{F}}') \end{array}}{\langle \mathcal{X} \rangle \Xi \vDash_3 \mathsf{CHOICE}\, (f' :: \mathcal{F}') \to (\overline{\mathrm{T}}, \langle \mathcal{X}_0 \rangle \Xi_0 \oplus x \mapsto \tau')}$$

$$\frac{\begin{array}{c} f' \lhd (l', [\,], \mathsf{TRef}\, x, \{\}, s') \\ \langle \mathcal{X} \rangle \Xi \vDash_3 \mathsf{CHOICE}\, \mathcal{F}' \to (\mathsf{CHOICE}\, \overline{\mathcal{F}}', \langle \mathcal{X}_0 \rangle \Xi_0) \qquad \overline{\mathrm{T}} \triangleq \mathsf{CHOICE}\, (f' :: \overline{\mathcal{F}}') \end{array}}{\langle \mathcal{X} \rangle \Xi \vDash_3 \mathsf{CHOICE}\, (f' :: \mathcal{F}') \to (\overline{\mathrm{T}}, \langle \mathcal{X}_0 \rangle \Xi_0)}$$

$$\frac{\begin{array}{c} \varphi' \lhd \mathsf{Field}\, (l', \tau', \mathsf{TRef}\, x, \{\}, s') \\ \tau' \not\lhd [\,] \qquad \langle \mathcal{X} \rangle \Xi \vDash_3 \mathsf{SET}\, \Phi' \to (\mathsf{SET}\, \overline{\Phi}', \langle \mathcal{X}_0 \rangle \Xi_0) \\ \overline{\mathrm{T}} \triangleq \mathsf{SET}\, (\mathsf{Field}\, (l', [\,], \mathsf{TRef}\, x, \{\}, s') :: \overline{\Phi}') \end{array}}{\langle \mathcal{X} \rangle \Xi \vDash_3 \mathsf{SET}\, (\varphi' :: \Phi') \to (\overline{\mathrm{T}}, \langle \mathcal{X}_0 \rangle \Xi_0 \oplus x \mapsto \tau')}$$

$$\frac{\begin{array}{c} \varphi' \lhd \mathsf{Field}\, (l', [\,], \mathsf{TRef}\, x, \{\}, s') \\ \langle \mathcal{X} \rangle \Xi \vDash_3 \mathsf{SET}\, \Phi' \to (\mathsf{SET}\, \overline{\Phi}', \langle \mathcal{X}_0 \rangle \Xi_0) \qquad \overline{\mathrm{T}} \triangleq \mathsf{SET}\, (\varphi' :: \overline{\Phi}') \end{array}}{\langle \mathcal{X} \rangle \Xi \vDash_3 \mathsf{SET}\, (\varphi' :: \Phi') \to (\overline{\mathrm{T}}, \langle \mathcal{X}_0 \rangle \Xi_0)}$$

$$\frac{\begin{array}{c} \varphi' \lhd \mathsf{Field}\, (l', \tau', \mathsf{TRef}\, x, \{\}, s') \\ \tau' \not\lhd [\,] \qquad \langle \mathcal{X} \rangle \Xi \vDash_3 \mathsf{SEQUENCE}\, \Phi' \to (\mathsf{SEQUENCE}\, \overline{\Phi}', \langle \mathcal{X}_0 \rangle \Xi_0) \\ \overline{\mathrm{T}} \triangleq \mathsf{SEQUENCE}\, (\mathsf{Field}\, (l', [\,], \mathsf{TRef}\, x, \{\}, s') :: \overline{\Phi}') \end{array}}{\langle \mathcal{X} \rangle \Xi \vDash_3 \mathsf{SEQUENCE}\, (\varphi' :: \Phi') \to (\overline{\mathrm{T}}, \langle \mathcal{X}_0 \rangle \Xi_0 \oplus x \mapsto \tau')}$$

$$\frac{\begin{array}{c} \varphi' \lhd \mathsf{Field}\, (l', [\,], \mathsf{TRef}\, x, \{\}, s') \\ \langle \mathcal{X} \rangle \Xi \vDash_3 \mathsf{SEQUENCE}\, \Phi' \to (\mathsf{SEQUENCE}\, \overline{\Phi}', \langle \mathcal{X}_0 \rangle \Xi_0) \\ \overline{\mathrm{T}} \triangleq \mathsf{SEQUENCE}\, (\varphi' :: \overline{\Phi}') \end{array}}{\langle \mathcal{X} \rangle \Xi \vDash_3 \mathsf{SEQUENCE}\, (\varphi' :: \Phi') \to (\overline{\mathrm{T}}, \langle \mathcal{X}_0 \rangle \Xi_0)}$$

$$\frac{\mathrm{T} \not\lhd \mathsf{SET\ OF\ \_\_\_ \mid SEQUENCE\ OF\ \_\_\_} \\ \mid \mathsf{CHOICE}\,(\_ :: \_) \mid \mathsf{SET}\,(\_ :: \_) \mid \mathsf{SEQUENCE}\,(\_ :: \_)}{\langle \mathcal{X} \rangle \Xi \vDash_3 \mathrm{T} \to (\mathrm{T}, \langle \mathcal{X} \rangle \Xi)}$$

Seconde étape : on enfile les étiquetages locaux récoltés précédemment, sur les types globaux.

$$\frac{x \in \mathcal{X} \qquad \Gamma(x) \lhd (\alpha, \tau, \{(\mathrm{T}, \sigma)\}) \\ \langle \mathcal{X} \rangle \Xi \vDash_4 \langle \mathcal{A} \rangle \Gamma \to \langle \overline{\mathcal{A}} \rangle \overline{\Gamma} \qquad \overline{\gamma} \triangleq x \mapsto (\alpha, \Xi(x) \mathbin{@} \tau, \{(\mathrm{T}, \sigma)\})}{\langle \mathcal{X} \rangle \Xi \vDash_4 \langle \{x\} \uplus \mathcal{A} \rangle \Gamma \to \langle \overline{\mathcal{A}} \rangle \overline{\Gamma} \oplus \overline{\gamma}}$$

$$\frac{x \notin \mathcal{X} \qquad \langle \mathcal{X} \rangle \Xi \vDash_4 \langle \mathcal{A} \rangle \Gamma \to \langle \overline{\mathcal{A}} \rangle \overline{\Gamma} \qquad \overline{\gamma} \triangleq x \mapsto \Gamma(x)}{\langle \mathcal{X} \rangle \Xi \vDash_4 \langle \{x\} \uplus \mathcal{A} \rangle \Gamma \to \langle \overline{\mathcal{A}} \rangle \overline{\Gamma} \oplus \overline{\gamma}}$$

$$\langle \mathcal{X} \rangle \Xi \vDash_4 \langle \varnothing \rangle \Gamma \to \langle \varnothing \rangle \Gamma$$

6.10.2 Ajout des étiquettes UNIVERSAL implicites

$$\frac{\Gamma(x) \lhd (\alpha, \tau, \{(\mathrm{T}, \sigma)\}) \\ \vDash_{\overline{32}} \mathrm{T} \to \overline{\tau} \qquad \vDash_{\overline{67}} \langle \mathcal{A} \rangle \Gamma \to \langle \overline{\mathcal{A}} \rangle \overline{\Gamma} \qquad \overline{\gamma} \triangleq x \mapsto (\alpha, \tau \mathbin{@} \overline{\tau}, \{(\mathrm{T}, \sigma)\})}{\vDash_{\overline{67}} \langle \{x\} \uplus \mathcal{A} \rangle \Gamma \to \langle \overline{\mathcal{A}} \rangle \overline{\Gamma} \oplus \overline{\gamma}}$$

$$\vDash_{\overline{67}} \langle \varnothing \rangle \Gamma \to \langle \varnothing \rangle \Gamma \qquad \frac{\overline{\tau} \triangleq [\mathsf{Tag}\ (\mathsf{UNIVERSAL}, \mathsf{Imm}\,1)\ \mathsf{IMPLICIT}]}{\vDash_{\overline{32}} \mathsf{BOOLEAN} \to \overline{\tau}}$$

$$\frac{\overline{\tau} \triangleq [\mathsf{Tag}\ (\mathsf{UNIVERSAL}, \mathsf{Imm}\,2)\ \mathsf{IMPLICIT}]}{\vDash_{\overline{32}} \mathsf{INTEGER}\ \ \_ \to \overline{\tau}}$$

$$\frac{\overline{\tau} \triangleq [\mathsf{Tag}\ (\mathsf{UNIVERSAL}, \mathsf{Imm}\,3)\ \mathsf{IMPLICIT}]}{\vDash_{\overline{32}} \mathsf{BIT\ STRING}\ \ \_ \to \overline{\tau}}$$

$$\frac{\overline{\tau} \triangleq [\mathsf{Tag}\ (\mathsf{UNIVERSAL}, \mathsf{Imm}\,4)\ \mathsf{IMPLICIT}]}{\vDash_{\overline{32}} \mathsf{OCTET\ STRING} \to \overline{\tau}}$$

$$\frac{\overline{\tau} \triangleq [\mathsf{Tag}\ (\mathsf{UNIVERSAL}, \mathsf{Imm}\,5)\ \mathsf{IMPLICIT}]}{\vDash_{\overline{32}} \mathsf{NULL} \to \overline{\tau}}$$

$$\frac{\overline{\tau} \triangleq [\mathsf{Tag}\ (\mathsf{UNIVERSAL}, \mathsf{Imm}\,6)\ \mathsf{IMPLICIT}]}{\vDash_{\overline{32}} \mathsf{OBJECT\ IDENTIFIER} \to \overline{\tau}}$$

$$\frac{\overline{\tau} \triangleq [\mathsf{Tag}\ (\mathsf{UNIVERSAL}, \mathsf{Imm}\, 10)\ \mathsf{IMPLICIT}]}{\vdash_{\overline{32}} \mathsf{ENUMERATED}\ \_ \to \overline{\tau}} \qquad \vdash_{\overline{32}} \mathsf{CHOICE}\ \_ \to [\,]$$

$$\frac{\overline{\tau} \triangleq [\mathsf{Tag}\ (\mathsf{UNIVERSAL}, \mathsf{Imm}\, 16)\ \mathsf{IMPLICIT}]}{\vdash_{\overline{32}} \mathsf{SEQUENCE\ OF}\ \_\ \_\ \_ \to \overline{\tau}}$$

$$\frac{\overline{\tau} \triangleq [\mathsf{Tag}\ (\mathsf{UNIVERSAL}, \mathsf{Imm}\, 16)\ \mathsf{IMPLICIT}]}{\vdash_{\overline{32}} \mathsf{SEQUENCE}\ \_ \to \overline{\tau}}$$

$$\frac{\overline{\tau} \triangleq [\mathsf{Tag}\ (\mathsf{UNIVERSAL}, \mathsf{Imm}\, 17)\ \mathsf{IMPLICIT}]}{\vdash_{\overline{32}} \mathsf{SET\ OF}\ \_\ \_\ \_ \to \overline{\tau}}$$

$$\frac{\overline{\tau} \triangleq [\mathsf{Tag}\ (\mathsf{UNIVERSAL}, \mathsf{Imm}\, 17)\ \mathsf{IMPLICIT}]}{\vdash_{\overline{32}} \mathsf{SET}\ \_ \to \overline{\tau}}$$

$$\frac{\overline{\tau} \triangleq [\mathsf{Tag}\ (\mathsf{UNIVERSAL}, \mathsf{Imm}\, 18)\ \mathsf{IMPLICIT}]}{\vdash_{\overline{32}} \mathsf{CharString}\,(\mathsf{Numeric}) \to \overline{\tau}}$$

$$\frac{\overline{\tau} \triangleq [\mathsf{Tag}\ (\mathsf{UNIVERSAL}, \mathsf{Imm}\, 19)\ \mathsf{IMPLICIT}]}{\vdash_{\overline{32}} \mathsf{CharString}\,(\mathsf{Printable}) \to \overline{\tau}}$$

$$\frac{\overline{\tau} \triangleq [\mathsf{Tag}\ (\mathsf{UNIVERSAL}, \mathsf{Imm}\, 20)\ \mathsf{IMPLICIT}]}{\vdash_{\overline{32}} \mathsf{CharString}\,(\mathsf{Teletex} \mid \mathsf{T}_{61}) \to \overline{\tau}}$$

$$\frac{\overline{\tau} \triangleq [\mathsf{Tag}\ (\mathsf{UNIVERSAL}, \mathsf{Imm}\, 21)\ \mathsf{IMPLICIT}]}{\vdash_{\overline{32}} \mathsf{CharString}\,(\mathsf{Videotex}) \to \overline{\tau}}$$

$$\frac{\overline{\tau} \triangleq [\mathsf{Tag}\ (\mathsf{UNIVERSAL}, \mathsf{Imm}\, 22)\ \mathsf{IMPLICIT}]}{\vdash_{\overline{32}} \mathsf{CharString}\,(\mathsf{IA}_5) \to \overline{\tau}}$$

$$\frac{\overline{\tau} \triangleq [\mathsf{Tag}\ (\mathsf{UNIVERSAL}, \mathsf{Imm}\, 25)\ \mathsf{IMPLICIT}]}{\vdash_{\overline{32}} \mathsf{CharString}\,(\mathsf{Graphic}) \to \overline{\tau}}$$

$$\frac{\overline{\tau} \triangleq [\mathsf{Tag}\ (\mathsf{UNIVERSAL}, \mathsf{Imm}\, 26)\ \mathsf{IMPLICIT}]}{\vdash_{\overline{32}} \mathsf{CharString}\,(\mathsf{Visible} \mid \mathsf{ISO}_{646}) \to \overline{\tau}}$$

$$\frac{\overline{\tau} \triangleq [\mathsf{Tag}\ (\mathsf{UNIVERSAL}, \mathsf{Imm}\, 27)\ \mathsf{IMPLICIT}]}{\vdash_{\overline{32}} \mathsf{CharString}\,(\mathsf{General}) \to \overline{\tau}}$$

6.10.3 Calcul des étiquetages

$$\frac{\Gamma(x) \lhd (\alpha, \tau, \{(\mathrm{T}, \sigma)\}) \qquad \langle\mathcal{B}\rangle\Delta \models_8 \tau \to \tau_0 \qquad \mathrm{T} \models_7 \tau_0 \to \overline{\tau} \qquad \langle\mathcal{B}\rangle\Delta \models_6 \langle\mathcal{A}\rangle\Gamma \to \langle\overline{\mathcal{A}}\rangle\overline{\Gamma} \qquad \overline{\gamma} \triangleq x \mapsto (\alpha, \overline{\tau}, \{(\mathrm{T}, \sigma)\})}{\langle\mathcal{B}\rangle\Delta \models_6 \langle\{x\} \uplus \mathcal{A}\rangle\Gamma \to \langle\overline{\mathcal{A}}\rangle\overline{\Gamma} \oplus \overline{\gamma}}$$

$$\langle\mathcal{B}\rangle\Delta \models_6 \langle\varnothing\rangle\Gamma \to \langle\varnothing\rangle\Gamma$$

$$\frac{y \in \mathcal{B} \qquad \Delta(y) \lhd ([\,], \mathsf{TRef}\ \_, \{\}, \mathsf{Int}\, n) \qquad \langle\mathcal{B}\rangle\Delta \models_8 (\mathsf{Tag}\ (c, \mathsf{Imm}\, n)\ a) :: \tau' \to \overline{\tau}}{\langle\mathcal{B}\rangle\Delta \models_8 (\mathsf{Tag}\ (c, \mathsf{Ref}\, y)\ a) :: \tau' \to \overline{\tau}}$$

$$\frac{t' \lhd (\mathsf{Tag}\ (c, \mathsf{Imm}\, n)\ a) \qquad n \geqslant 0 \qquad \langle\mathcal{B}\rangle\Delta \models_8 \tau' \to \overline{\tau}'}{\langle\mathcal{B}\rangle\Delta \models_8 t' :: \tau' \to t' :: \overline{\tau}'}$$

$$\langle\mathcal{B}\rangle\Delta \models_8 [\,] \to [\,] \qquad\qquad \mathrm{T} \models_7 [\,] \to [\,]$$

$$\frac{\mathrm{T} \models_7 \tau \to \overline{\tau}}{\mathrm{T} \models_7 (\mathsf{Tag}\ \psi\ \mathsf{EXPLICIT}) :: \tau \to (\mathsf{Tag}\ \psi\ \mathsf{EXPLICIT}) :: \overline{\tau}}$$

$$\frac{\mathrm{T} \not\lhd \mathsf{CHOICE}\ \_}{\mathrm{T} \models_7 [(\mathsf{Tag}\ \psi\ \mathsf{IMPLICIT})] \to [(\mathsf{Tag}\ \psi\ \mathsf{IMPLICIT})]}$$

$$\frac{\tau \not\lhd [\,] \qquad \mathrm{T} \models_7 \tau \to \_ :: \overline{\tau}}{\mathrm{T} \models_7 (\mathsf{Tag}\ \psi\ \mathsf{IMPLICIT}) :: \tau \to (\mathsf{Tag}\ \psi\ \mathsf{EXPLICIT}) :: \overline{\tau}}$$

$$\frac{\mathrm{default\_tagging} \lhd \mathsf{None} \mid \mathsf{Some}\,(\mathsf{EXPLICIT\ TAGS}) \qquad \mathrm{T} \models_7 \tau \to \overline{\tau}}{\mathrm{T} \models_7 (\mathsf{Tag}\ \psi\ \mathsf{Inferred}) :: \tau \to (\mathsf{Tag}\ \psi\ \mathsf{EXPLICIT}) :: \overline{\tau}}$$

$$\mathsf{CHOICE}\ \_ \models_7 [\mathsf{Tag}\ \psi\ \mathsf{Inferred}] \to [\mathsf{Tag}\ \psi\ \mathsf{EXPLICIT}]$$

$$\frac{\mathrm{default\_tagging} \lhd \mathsf{Some}\,(\mathsf{IMPLICIT\ TAGS} \mid \mathsf{AUTOMATIC\ TAGS}) \qquad \tau \not\lhd [\,] \qquad \mathrm{T} \models_7 \tau \to \_ :: \overline{\tau}'}{\mathrm{T} \models_7 (\mathsf{Tag}\ \psi\ \mathsf{Inferred}) :: \tau \to (\mathsf{Tag}\ \psi\ \mathsf{EXPLICIT}) :: \overline{\tau}'}$$

6.10.4 Normalisation des étiquetages

$$\frac{\begin{array}{c}\langle\varnothing\rangle\Xi \vdash_{\overline{49}} \langle\mathcal{A}\rangle\Gamma \to (\langle\mathcal{A}_0\rangle\Gamma_0, \langle\mathcal{X}_0\rangle\Xi_0) \qquad \langle\mathcal{X}_0\rangle\Xi_0 \vdash_{\overline{4}} \langle\mathcal{A}_0\rangle\Gamma_0 \to \langle\mathcal{A}_1\rangle\Gamma_1 \\ \vdash_{\overline{67}} \langle\mathcal{A}_1\rangle\Gamma_1 \to \langle\mathcal{A}_2\rangle\Gamma_2 \qquad \langle\mathcal{B}\rangle\Delta \vdash_{\overline{6}} \langle\mathcal{A}_2\rangle\Gamma_2 \to \langle\overline{\mathcal{A}}\rangle\overline{\Gamma}\end{array}}{\langle\mathcal{B}\rangle\Delta \vdash_{\overline{33}} \langle\mathcal{A}\rangle\Gamma \to \langle\overline{\mathcal{A}}\rangle\overline{\Gamma}}$$

6.11 Définition des environnements canoniques

$$\frac{\begin{array}{c}\vdash_{\overline{25}} \langle\mathcal{A}\rangle\Gamma \to \langle\mathcal{A}_0\rangle\Gamma_0 \\ \Vdash_{\overline{3}} \langle\mathcal{A}_0\rangle\Gamma_0 \qquad \vdash_{\overline{10}} (\langle\mathcal{A}_0\rangle\Gamma_0, \langle\mathcal{B}\rangle\Delta) \to (\langle\mathcal{A}_1\rangle\Gamma_1, \langle\mathcal{B}_0\rangle\Delta_0) \\ \vdash_{\overline{31}} \langle\mathcal{A}_1\rangle\Gamma_1 \to \langle\mathcal{A}_2\rangle\Gamma_2 \qquad \langle\mathcal{B}_0\rangle\Delta_0 \vdash_{\overline{48}} \langle\mathcal{A}_2\rangle\Gamma_2 \to (\langle\mathcal{A}_3\rangle\Gamma_3, \langle\mathcal{B}_1\rangle\Delta_1) \\ \langle\mathcal{B}_1\rangle\Delta_1 \vdash_{\overline{53}} \langle\mathcal{A}_3\rangle\Gamma_3 \to (\langle\mathcal{A}_4\rangle\Gamma_4, \langle\mathcal{B}_2\rangle\Delta_2) \qquad \langle\mathcal{A}_4\rangle\Gamma_4 \vdash_{\overline{18}} \langle\mathcal{B}_2\rangle\Delta_2 \to \langle\overline{\mathcal{B}}\rangle\overline{\Delta} \\ \langle\overline{\mathcal{B}}\rangle\overline{\Delta} \vdash_{\overline{96}} \langle\mathcal{A}_4\rangle\Gamma_4 \to \langle\mathcal{A}_5\rangle\Gamma_5 \qquad \vdash_{\overline{30}} \langle\mathcal{A}_5\rangle\Gamma_5 \qquad \langle\overline{\mathcal{B}}\rangle\overline{\Delta} \vdash_{\overline{33}} \langle\mathcal{A}_5\rangle\Gamma_5 \to \langle\overline{\mathcal{A}}\rangle\overline{\Gamma}\end{array}}{\vdash_{\overline{47}} (\langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta) \to (\langle\overline{\mathcal{A}}\rangle\overline{\Gamma}, \langle\overline{\mathcal{B}}\rangle\overline{\Delta})}$$

Chapitre 7

Contrôle des types

7.1 Définition

Nous présentons ici le prédicat de contrôle des types qui, étant donné une valeur canonique, un environnement de types canoniques et un type de cet environnement, décide si la valeur est du type donné dans l'environnement.

Pour suivre la présentation il n'est pas nécessaire d'avoir compris ni lu entièrement le chapitre consacré à l'obtention des environnements canoniques : nous supposerons, comme Leibniz, que nous vivons dans le meilleur des environnements possibles, et pour les ironiques Zadig nous indiquerons à chaque fois ce qui est implicitement supposé.

Le prédicat a la forme : $\langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathrm{T}$ Notons la présence dans le contexte d'un champ x. Il a pour type string. Il désigne le nom du type T dans l'environnement $\langle\mathcal{A}\rangle\Gamma$ (c'est-à-dire $x \in \mathcal{A}$). Il est nécessaire à cause du type REAL (voir section 6.1 page 119). En effet celui-ci est défini dans cette thèse de la façon suivante :

```
REAL ::= SEQUENCE {
           mantissa INTEGER,
           base     INTEGER (2 | 10),
           exponent INTEGER
         }
```

Nous évitons ainsi l'écueil de la relation d'association, non spécifiée mais employée dans (ITU, 1994a, § 18.5). REAL reste un mot-clé. Trois valeurs de ce type ne sont cependant pas exprimables à l'aide de la définition ci-dessus : 0.0, MINUS-INFINITY et PLUS-INFINITY (ITU, 1994a, § 18.6). Rappelons que 0.0 est l'arbre de syntaxe abstraite canonique mis pour l'extrait de syntaxe concrète 0 qui est ambigu car il peut être interprété à la fois comme relevant du type REAL et INTEGER. En ce sens la cano-

nisation, et en particulier à la section 6.7.1 page 132, a déjà effectué une partie du contrôle des types.

Le champ contextuel x de la relation de contrôle des types sert alors à traiter spécialement les cas des valeurs spéciales du type REAL : 0.0, MINUS-INFINITY et PLUS-INFINITY.

$$\frac{x' \in \mathcal{A} \qquad \Gamma(x') \lhd (\alpha', \tau', \{(\mathrm{T}', \sigma')\}) \qquad \langle\mathcal{A}\rangle\Gamma, x' \vdash v : \mathrm{T}'}{\langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathsf{TRef}\,(x')} \text{ [1]}$$

C'est la règle de dépliage du type. Si celui-ci est une abréviation (donc de la forme $\mathsf{TRef}\,(x')$) valide (c'est-à-dire que x' est le nom d'un type dans l'environnement : $x' \in \mathcal{A}$), alors on extrait de l'environnement l'entrée référencée ($\Gamma(x') \lhd (\alpha', \tau', \{(\mathrm{T}', \sigma')\})$) et on produit un nouveau jugement avec le nouveau type, soit, $\langle\mathcal{A}\rangle\Gamma, x' \vdash v : \mathrm{T}'$.

$$\langle\mathcal{A}\rangle\Gamma, x \vdash (\mathsf{FALSE} \mid \mathsf{TRUE}) : \mathsf{BOOLEAN} \text{ [2]}$$

Les seules valeurs de type BOOLEAN sont FALSE et TRUE.

$$\langle\mathcal{A}\rangle\Gamma, x \vdash \mathsf{NULL} : \mathsf{NULL} \text{ [3]}$$

C'est le cas du type NULL dont l'unique valeur est NULL. Attention ! Dans la syntaxe abstraite, NULL apparaît à la fois comme valeur et type ASN.1 (sections 5.3 page 101 et 5.5 page 111), pour refléter la surcharge de la norme (ITU, 1994a, § 21).

$$\frac{v \lhd \mathsf{MINUS\text{-}INFINITY} \mid \mathsf{PLUS\text{-}INFINITY}}{\langle\mathcal{A}\rangle\Gamma, \text{"REAL"} \vdash v : \mathrm{T}} \text{ [4]}$$

Nous traitons ici le cas de deux des trois valeurs spéciales du type REAL. On notera l'usage du champ x au lieu de T qui vaut SEQUENCE ...

$$\frac{\text{String.length}\,(s) \equiv 0 \text{ [modulo 8]}}{\langle\mathcal{A}\rangle\Gamma, x \vdash \mathsf{BinStr}\,(s) : \mathsf{OCTET\ STRING}} \text{ [5]}$$

Les chaînes numériques ont été normalisées lors de la canonisation des environnements (section 6.7.2 page 134). Elles sont toutes sous la forme de chaînes de bits (BinStr). Pour savoir si elles sont bien du type OCTET STRING il suffit alors de vérifier que leur longueur est un multiple de 8. String.length est la fonction OCaml pour calculer la longueur d'une chaîne de caractères.

$$\langle\mathcal{A}\rangle\Gamma, x \vdash \mathsf{Int\_} : \mathsf{INTEGER} \quad \_ \text{ [6]}$$

Cette règle dit que toutes les valeurs syntaxiques de constructeur Int sont de type INTEGER. Il est important de noter que le cas de l'extrait de

syntaxe concrète 0 a été réécrit en l'arbre de syntaxe abstraite Int (0) lors de la canonisation.

$$\langle \mathcal{A} \rangle \Gamma, \text{"REAL"} \vdash 0.0 : \mathrm{T} \ [7]$$

Cette règle traite le cas de la valeur spéciale 0.0 de type REAL. À noter que nous ne projetons pas le terme T, mais le champ contextuel x.

$$\langle \mathcal{A} \rangle \Gamma, x \vdash \mathsf{Str\_} : \mathsf{CharString}\ \_ \ [8]$$

Nous traitons ici le cas de toutes les chaînes de caractères (proprement dites). Nous n'effectuons pas les vérifications spécifiées en (ITU, 1994a, § 34), car celles-ci sont dépourvues de difficulté et ne sont d'aucun intérêt dans l'optique de cette thèse.

$$\langle \mathcal{A} \rangle \Gamma, x \vdash \mathsf{BinStr}\ \_ : \mathsf{BIT\ STRING}\ \ \_ \ [9]$$

Cette règle est complémentaire de la règle [5]. Nous acceptons toutes les chaînes de bits comme étant de type BIT STRING.

$$\frac{\mathrm{T} \lhd \mathsf{ENUMERATED}\ ([(y, \_)] \sqcup \mathcal{C}')}{\langle \mathcal{A} \rangle \Gamma, x \vdash \mathsf{Enum}\,(y) : \mathrm{T}} \ [10]$$

Cette règle spécifie qu'une valeur annoncée de constructeur Enum doit référencer une des constantes énumératives du type ENUMERATED pour être de ce type. À noter que les dés sont un peu pipés car le constructeur de valeurs Enum a été introduit dans l'arbre de syntaxe abstraite lors de la normalisation de valeurs, et non à l'analyse syntaxique (section 6.7.2 page 134).

$$\frac{\begin{array}{c}\mathrm{T} \lhd (\mathsf{SEQUENCE\ OF} \mid \mathsf{SET\ OF})\ \tau'\,\mathrm{T}'\,\sigma' \\ \langle \mathcal{A} \rangle \Gamma, \text{""} \vdash v' : \mathrm{T}' \qquad \langle \mathcal{A} \rangle \Gamma, x \vdash \{V'\} : \mathrm{T}\end{array}}{\langle \mathcal{A} \rangle \Gamma, x \vdash \{[(\mathsf{None}, v')] \sqcup V'\} : \mathrm{T}} \ [11]$$

$$\frac{\mathrm{T} \lhd (\mathsf{SEQUENCE\ OF} \mid \mathsf{SET\ OF})\ \tau'\,\mathrm{T}'\,\sigma'}{\langle \mathcal{A} \rangle \Gamma, x \vdash \{[\,]\} : \mathrm{T}} \ [12]$$

Ces règles traitent le cas des types SET OF et SEQUENCE OF. La règle [12] est appliquée si la valeur est $\{[\,]\}$ (en syntaxe concrète : {}). Considérons la règle [11] en détail. Nous avons : $v \lhd \{[(\mathsf{None}, v')] \sqcup V'\}$. Le symbole $\sqcup$ dénote la fonction d'union de deux listes (l'entrelacement des éléments n'est pas spécifié). Donc ici cela spécifie que l'ordre dans lequel sont examinées les valeurs v' n'est pas significatif. La seconde prémisse établit le contrôle du type d'une valeur v'. À noter d'ailleurs la valeur ""

du champ x : ceci parce que le type T est sous forme canonique, et donc ses champs sont en fait constitués d'abréviations. Par conséquent la règle de dépliage [1] s'appliquera à T' et donnera une valeur pertinente à ce champ. La dernière prémisse s'interprète algorithmiquement comme l'appel récursif, pour traiter les valeurs éventuellement restantes.

$$\frac{f' \lhd (l', \tau', \mathrm{T}', \sigma', s') \qquad \langle\mathcal{A}\rangle\Gamma, x = \text{""} \vdash v' : \mathrm{T}'}{\langle\mathcal{A}\rangle\Gamma, x \vdash (l' : v') : \mathsf{CHOICE}\,([f'] \sqcup \mathcal{F}')} \text{ [13]}$$

$$\frac{\langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathsf{CHOICE}\,\mathcal{F}'}{\langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathsf{CHOICE}\,([f'] \sqcup \mathcal{F}')} \text{ [14]}$$

Ces règles traitent le cas où $\mathrm{T} \lhd$ CHOICE _. La règle [14] spécifie qu'il est possible d'ignorer une variante (c'est-à-dire un champ de CHOICE). La règle [13] traite le cas où l'on projette le CHOICE pour en sélectionner une variante. La première prémisse $f' \lhd (l', \tau', \mathrm{T}', \sigma', s')$ impose que le label de cette variante doit être identique à l'identificateur contenu dans la valeur l'. La seconde prémisse projette la valeur et le type et contrôle leur adéquation. On notera dans cette prémisse la valeur spéciale du champ x (qui vaut ""), due au fait que l'environnement est canonique, et donc que T est en réalité une abréviation, ce qui ne rend pas pertinent ce champ x (voir les règles [11] et [12]). D'autre part, pour bien comprendre ces règles, comme pour bien comprendre les cas $\mathrm{T} \lhd$ SET _ | SEQUENCE _, il faut savoir que la canonicité des types n'assure pas que les labels (i.e. les noms des champs) soient distincts deux à deux. Ainsi il est tout à fait permis ici d'avoir un arbre de syntaxe abstraite correspondant au fragment de syntaxe concrète :

```
T ::= CHOICE {
        a INTEGER,
        a INTEGER,
        b BOOLEAN
      }
```

Pourquoi ce choix, alors que (ITU, 1994a, § 26.6) interdit cela ? Tout d'abord parce que du point de vue du contrôle de type, cette restriction n'est pas nécessaire, comme nous allons le vérifier. Ainsi la définition des environnements canoniques en est allégée. De plus, cette propriété d'unicité des labels sera définie ultérieurement et mise en parallèle avec celle d'unicité des étiquetages, dans le chapitre consacré à la sémantique (section 8 page 159), ce qui donnera une nouvelle perspective à la raison d'être fondamentale de cette restriction. Il est donc permis d'avoir :

```
v T ::= a : 7
```

En effet, pour prouver le jugement associé

$$\langle\mathcal{A}\rangle\Gamma, x \vdash (\text{"a"} : \mathsf{Int}\,(7)) : \mathrm{T}$$

où $\mathrm{T} \lhd \mathsf{CHOICE}\ \_$, nous avons alors le choix d'appliquer d'abord (nous raisonnons en partant de la racine de l'arbre de preuve) la règle [13] ou [14]. Si nous appliquons d'abord la règle [13], nous devons alors prouver :

$$\langle \mathcal{A} \rangle \Gamma, \text{""} \vdash \mathsf{Int}\,(7) : \mathsf{INTEGER}\,[\,]$$

ce qui est vrai par la règle [6]. Si en revanche nous appliquons la règle [14] d'abord, alors nous devons contrôler

```
v CHOICE {
    a INTEGER,
    b BOOLEAN
  } ::= a : 7
```

ce qui sera fait ensuite en appliquant la règle [13], puis [6].

On remarque que ni [13] ni [14] ne filtrent $\mathsf{CHOICE}\,[\,]$.

$$\frac{\begin{array}{cc}\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') & V \lhd [(\mathsf{Some}\, l', v')] \sqcup V' \\ \langle \mathcal{A} \rangle \Gamma, \text{""} \vdash v' : \mathrm{T}' & \langle \mathcal{A} \rangle \Gamma, x \vdash \{V'\} : \mathsf{SEQUENCE}\,\Phi'\end{array}}{\langle \mathcal{A} \rangle \Gamma, x \vdash \{V\} : \mathsf{SEQUENCE}\,([\varphi'] \sqcup \Phi')}\ [15]$$

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \\ s' \lhd \mathsf{Some}\,\mathsf{OPTIONAL} \qquad \langle \mathcal{A} \rangle \Gamma, x \vdash v : \mathsf{SEQUENCE}\,\Phi'\end{array}}{\langle \mathcal{A} \rangle \Gamma, x \vdash v : \mathsf{SEQUENCE}\,([\varphi'] \sqcup \Phi')}\ [16]$$

$$\langle \mathcal{A} \rangle \Gamma, x \vdash \{[\,]\} : \mathsf{SEQUENCE}\,[\,]\ [17]$$

Ces règles traitent le cas où $\mathrm{T} \lhd \mathsf{SEQUENCE}\ \_$. La règle [17] est le sous-cas de la séquence vide (ITU, 1994a, § 22.13.b). À ce sujet, notons que le cas (ITU, 1994a, § 22.13.a), c'est-à-dire par exemple :

```
v SEQUENCE {a REAL OPTIONAL} ::= {}
```

n'est pas prouvable directement par cette règle [17] : il faut d'abord appliquer la règle [16] pour éliminer le champ optionnel. La règle [16] stipule en effet qu'un champ optionnel peut être ignoré. Par définition, il n'est pas nécessaire que la séquence comporte alors une valeur associée à ce champ (conformément à (ITU, 1994a, § 22.8). Remarquons que le cas où un champ est marqué DEFAULT n'apparaît pas ici, mais est géré par la règle [15], au même titre que les champs obligatoires. Ceci est dû au fait que les valeurs sont canoniques, et donc en particulier les valeurs par défaut, si besoin, ont été introduites dans la valeur séquence (nous accomplissons ainsi (ITU, 1994a, § 22.9). La règle [15] traite le cas où une

valeur de la séquence est confrontée au type d'un champ de SEQUENCE. Le fait que l'on ait :

$$\begin{gathered}\mathrm{T} \lhd \mathsf{SEQUENCE}\,([\varphi'] \sqcup \Phi') \\ \varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \\ V \lhd [(\mathsf{Some}\,l', v')] \sqcup V'\end{gathered}$$

signifie que l'on peut choisir parmi les valeurs de la séquence une qui ait pour nom un label du type SEQUENCE, la façon dont on fait ces deux choix n'étant pas spécifiée. Rappelons à ce sujet que $\sqcup$, contrairement à l'union ensembliste $\cup$, est une union disjointe (ainsi si $\Phi \lhd [\varphi'] \sqcup \Phi'$ alors Φ' est une liste strictement plus courte que Φ.)

On notera que, comme pour le cas CHOICE (cf. règles [13] et [14]), il n'est pas imposé ici que les labels soient distincts deux à deux (comme le demande (ITU, 1994a, § 22.10)) : ce n'est pas du ressort du contrôle de type à proprement parler, mais de la canonisation des types.

D'autre part, on remarquera que nous n'imposons pas que les valeurs de la séquence soient dans l'ordre des champs du type SEQUENCE, comme cela est pourtant exigé par (ITU, 1994a, § 22.14). Nous considérons en effet que cela n'est pas du ressort du contrôle des types, aussi la vérification de cette exigence a été faite lors de la canonisation des types. Nous soutenons par ailleurs que cette contrainte est inutile dans la sémantique statique, car la seule différence entre SEQUENCE et SET réside dans le codage de leurs valeurs (section 8.1 page 159) et le contrôle sémantique des types (section 8.2 page 161) — que l'on peut assimiler à un décodage.

$$\frac{\langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathsf{SEQUENCE}\,\Phi}{\langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathsf{SET}\,\Phi}\ [18]$$

Le cas du type SET est traité exactement comme celui du type SEQUENCE (cf. ci-dessus pour une explication).

7.2 Types bien labellisés

$$\frac{\langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{L}} \mathrm{T}}{\langle\mathcal{A}\rangle\Gamma \vdash_{\mathcal{L}} \mathrm{T}}\ [1]$$

$$\frac{x' \in \mathcal{A} \qquad \Gamma(x') \lhd (\alpha', \tau', \{(\mathrm{T}', \sigma')\}) \qquad \langle\mathcal{A}\rangle\Gamma, \mathrm{L} \vdash_{\mathcal{L}} \mathrm{T}'}{\langle\mathcal{A}\rangle\Gamma, \mathrm{L} \vdash_{\mathcal{L}} \mathsf{TRef}\,(x')}\ [2]$$

$$\frac{f' \lhd (l', \tau', \mathrm{T}', \sigma', s') \qquad \langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{L}} \mathrm{T}' \qquad \langle\mathcal{A}\rangle\Gamma, \{l'\} \uplus \mathrm{L} \vdash_{\mathcal{L}} \mathsf{CHOICE}\,\mathcal{F}'}{\langle\mathcal{A}\rangle\Gamma, \mathrm{L} \vdash_{\mathcal{L}} \mathsf{CHOICE}\,([f'] \sqcup \mathcal{F}')}\ [3]$$

$$\frac{\langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{L}} \mathrm{T}'}{\langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{L}} (\mathsf{SEQUENCE\ OF} \mid \mathsf{SET\ OF})\, \tau'\, \mathrm{T}'\, \sigma'} \ [4]$$

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \\ \langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{L}} \mathrm{T}' \qquad \langle\mathcal{A}\rangle\Gamma, \{l'\} \uplus \mathrm{L} \vdash_{\mathcal{L}} \mathsf{SEQUENCE}\,\Phi'\end{array}}{\langle\mathcal{A}\rangle\Gamma, \mathrm{L} \vdash_{\mathcal{L}} \mathsf{SEQUENCE}\,([\varphi'] \sqcup \Phi')} \ [5]$$

$$\frac{\langle\mathcal{A}\rangle\Gamma, \mathrm{L} \vdash_{\mathcal{L}} \mathsf{SEQUENCE}\,\Phi}{\langle\mathcal{A}\rangle\Gamma, \mathrm{L} \vdash_{\mathcal{L}} \mathsf{SET}\,\Phi} \ [6]$$

$$\frac{\begin{array}{c}\mathrm{T} \not\lhd \mathsf{TRef}\ \_ \mid \mathsf{CHOICE}\ \_ \\ \mid \mathsf{SET\ OF}\ \_\_\_ \mid \mathsf{SEQUENCE\ OF}\ \_\_\_ \\ \mid \mathsf{SET}\ \_ \mid \mathsf{SEQUENCE}\ \_\end{array}}{\langle\mathcal{A}\rangle\Gamma, \mathrm{L} \vdash_{\mathcal{L}} \mathrm{T}} \ [7]$$

7.3 Unicité du contrôle des types

Théorème 7.3.1 (Unicité du contrôle des types)**.**

> S'il existe une preuve qu'une valeur est d'un type bien labellisé, alors cette preuve est unique.

Nous avons d'abord besoin de deux lemmes techniques. Pour la preuve du théorème, cf. 7.3 page 155.

Lemme 7.3.2.

Si	$\langle\mathcal{A}\rangle\Gamma, L \vdash_{\mathcal{L}} \mathit{SEQUENCE}\,\Phi$
et	$\langle\mathcal{A}\rangle\Gamma, x \vdash \{V\} : \mathit{SEQUENCE}\,\Phi$
alors	$\forall l \in L.\forall(\mathsf{Some}\,x', \_) \in V.x' \neq l$

Preuve du lemme 7.3.2

La démonstration se fera par induction.

— $\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s')$ et $\Phi \lhd [\varphi'] \sqcup \Phi'$ et $V \lhd [(\mathsf{Some}\,l', v')] \sqcup V'$.
Distinguons une des hypothèses de filtre :
— **HF.0** : $V \lhd [(\mathsf{Some}\,l', v')] \sqcup V'$
Les autres hypothèses sont

— **H.0** : $\langle\mathcal{A}\rangle\Gamma, \mathrm{L} \vdash_{\mathcal{Z}} \mathsf{SEQUENCE}\,\Phi$
C'est la règle [5] (page 149) qui s'applique :

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \\ \langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{Z}} \mathrm{T}' \qquad \langle\mathcal{A}\rangle\Gamma, \{l'\} \uplus \mathrm{L} \vdash_{\mathcal{Z}} \mathsf{SEQUENCE}\,\Phi'\end{array}}{\langle\mathcal{A}\rangle\Gamma, \mathrm{L} \vdash_{\mathcal{Z}} \mathsf{SEQUENCE}\,([\varphi'] \sqcup \Phi')}\ [5]$$

Appliquons-là directement (i.e. toutes les variables de la règle sont liées aux variables de même nom dans l'environnement du cas courant de la preuve). Étiquetons ses prémisses non redondantes avec les hypothèses de filtre :
— **H.0.0** : $\langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{Z}} \mathrm{T}'$
— **H.0.1** : $\langle\mathcal{A}\rangle\Gamma, \{l'\} \uplus \mathrm{L} \vdash_{\mathcal{Z}} \mathsf{SEQUENCE}\,\Phi'$
— **H.1** : $\langle\mathcal{A}\rangle\Gamma, x \vdash \mathbf{\{}V\mathbf{\}} : \mathsf{SEQUENCE}\,\Phi$
HF.0 nous permet d'appliquer la règle [15] (page 147) :

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \qquad V \lhd [(\mathsf{Some}\,l', v')] \sqcup V' \\ \langle\mathcal{A}\rangle\Gamma, "" \vdash v' : \mathrm{T}' \qquad \langle\mathcal{A}\rangle\Gamma, x \vdash \mathbf{\{}V'\mathbf{\}} : \mathsf{SEQUENCE}\,\Phi'\end{array}}{\langle\mathcal{A}\rangle\Gamma, x \vdash \mathbf{\{}V\mathbf{\}} : \mathsf{SEQUENCE}\,([\varphi'] \sqcup \Phi')}\ [15]$$

Appliquons-là directement et étiquetons ses prémisses non redondantes :
— **H.1.0** : $\langle\mathcal{A}\rangle\Gamma, "" \vdash v' : \mathrm{T}'$
— **H.1.1** : $\langle\mathcal{A}\rangle\Gamma, x \vdash \mathbf{\{}V'\mathbf{\}} : \mathsf{SEQUENCE}\,\Phi'$
Prouvons
— **C** : $\forall l \in \mathrm{L}.\forall(\mathsf{Some}\,x', \_) \in V.x' \neq l$
Reprenons l'énoncé du théorème et effectuons-y les renommages suivants :
— $\Phi \xleftarrow{\alpha} \Phi'$
— $\mathrm{L} \xleftarrow{\alpha} \{l'\} \uplus \mathrm{L}$
— $V \xleftarrow{\alpha} V'$

Si	$\langle\mathcal{A}\rangle\Gamma, \{l'\} \uplus \mathrm{L} \vdash_{\mathcal{Z}} \mathsf{SEQUENCE}\,\Phi'$
et	$\langle\mathcal{A}\rangle\Gamma, x \vdash \mathbf{\{}V'\mathbf{\}} : \mathsf{SEQUENCE}\,\Phi'$
alors	$\forall l \in \{l'\} \uplus \mathrm{L}.\forall(\mathsf{Some}\,x', \_) \in V'.x' \neq l$

Nous reconnaissons en cet énoncé une instance de l'hypothèse d'induction, car ses prémisses sont, dans l'ordre d'écriture, **H.0.1** et **H.1.1**. En l'appliquant il vient donc :
— **H.2** : $\forall l \in \{l'\} \uplus \mathrm{L}.\forall(\mathsf{Some}\,x', \_) \in V'.x' \neq l$
H.2 implique
— **H.3** : $\forall l \in \mathrm{L}.\forall\mathsf{Some}\,x', \_) \in V'.x' \neq l$
— **H.4** : $\forall l \in \mathrm{L}.l' \neq l$
H.3 et **H.4** impliquent

— **H.5** : $\forall l \in \mathrm{L}.(l' \neq l \wedge \forall(\mathsf{Some}\, x', \_) \in V'.x' \neq l)$
H.5 et **HF.0** impliquent **C**. $\triangle$
— $\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s')$
et $\Phi \lhd [\varphi'] \sqcup \Phi'$
et $v \lhd \{V\}$
et $s' \lhd \mathsf{Some\ OPTIONAL}$

Distinguons deux des hypothèses de filtre :
— **HF.1** : $s' \lhd \mathsf{Some\ OPTIONAL}$
Les autres hypothèses sont
— **H.0** : $\langle\mathcal{A}\rangle\Gamma, \mathrm{L} \vdash_{\mathcal{L}} \mathsf{SEQUENCE}\,\Phi$
C'est la règle [5] (page 149) qui s'applique :

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \\ \langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{L}} \mathrm{T}' \qquad \langle\mathcal{A}\rangle\Gamma, \{l'\} \uplus \mathrm{L} \vdash_{\mathcal{L}} \mathsf{SEQUENCE}\,\Phi'\end{array}}{\langle\mathcal{A}\rangle\Gamma, \mathrm{L} \vdash_{\mathcal{L}} \mathsf{SEQUENCE}\,([\varphi'] \sqcup \Phi')}\ [5]$$

Appliquons-là directement et étiquetons ses prémisses non redondantes avec les hypothèses de filtre :
— **H.0.0** : $\langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{L}} \mathrm{T}'$
— **H.0.1** : $\langle\mathcal{A}\rangle\Gamma, \{l'\} \uplus \mathrm{L} \vdash_{\mathcal{L}} \mathsf{SEQUENCE}\,\Phi'$
— **H.1** : $\langle\mathcal{A}\rangle\Gamma, x \vdash \{V\} : \mathsf{SEQUENCE}\,\Phi$
La règle [16] (page 147) peut s'appliquer :

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \\ s' \lhd \mathsf{Some\,OPTIONAL} \qquad \langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathsf{SEQUENCE}\,\Phi'\end{array}}{\langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathsf{SEQUENCE}\,([\varphi'] \sqcup \Phi')}\ [16]$$

Appliquons-là en effectuant la substitution
— $v \xleftarrow{\beta} \{V\}$
Étiquetons sa prémisse non redondante :
— **H.1.0** : $\langle\mathcal{A}\rangle\Gamma, x \vdash \{V\} : \mathsf{SEQUENCE}\,\Phi'$
Prouvons
— **C** : $\forall l \in \mathrm{L}.\forall(\mathsf{Some}\, x', \_) \in V.x' \neq l$
Reprenons l'énoncé du théorème et effectuons-y les renommages suivants :
— $\Phi \xleftarrow{\alpha} \Phi'$
— $\mathrm{L} \xleftarrow{\alpha} \{l'\} \uplus \mathrm{L}$

Si	$\langle\mathcal{A}\rangle\Gamma, \{l'\} \uplus \mathrm{L} \vdash_{\mathcal{L}} \mathsf{SEQUENCE}\,\Phi'$
et	$\langle\mathcal{A}\rangle\Gamma, x \vdash \{V\} : \mathsf{SEQUENCE}\,\Phi'$
alors	$\forall l \in \{l'\} \uplus \mathrm{L}.\forall(\mathsf{Some}\, x', \_) \in V.x' \neq l$

Nous reconnaissons en cet énoncé une instance de l'hypothèse

d'induction, car ses prémisses sont, dans l'ordre d'écriture, **H.0.1** et **H.1.0**. En l'appliquant il vient donc :
— **H.2** : $\forall l \in \{l'\} \uplus \mathrm{L}.\forall(\mathsf{Some}\, x', \_) \in V.x' \neq l$
Puis **H.2** implique trivialement **C**. △
— $\Phi \lhd [\,]$
et $V \lhd [\,]$
et $v \lhd \mathbf{\{}V\mathbf{\}}$

Distinguons des hypothèses de filtre :
— **HF.0** : $\Phi \lhd [\,]$
— **HF.1** : $V \lhd [\,]$
Les autres hypothèses sont
— **H.0** : $\langle\mathcal{A}\rangle\Gamma, \mathrm{L} \vdash_{\mathcal{Z}} \mathsf{SEQUENCE}\,\Phi$
HF.0 et **HF.1** font que c'est la règle [7] (page 149) qui s'applique :

$$\frac{\begin{array}{c}\mathrm{T} \not\lhd \mathsf{TRef}\ \_ \mid \mathsf{CHOICE}\ \_ \\ \mid \mathsf{SET\ OF}\ \_\_\_ \mid \mathsf{SEQUENCE\ OF}\ \_\_\_ \\ \mid \mathsf{SET}\ \_ \mid \mathsf{SEQUENCE}\ \_\end{array}}{\langle\mathcal{A}\rangle\Gamma, \mathrm{L} \vdash_{\mathcal{Z}} \mathrm{T}}\ [7]$$

— **H.1** : $\langle\mathcal{A}\rangle\Gamma, x \vdash \mathbf{\{}V\mathbf{\}} : \mathsf{SEQUENCE}\,\Phi$
HF.0 et **HF.1** font que c'est la règle [17] (page 147) qui s'applique :

$$\langle\mathcal{A}\rangle\Gamma, x \vdash \mathbf{\{}[\,]\mathbf{\}} : \mathsf{SEQUENCE}\,[\,]\ [17]$$

Prouvons
— **C** : $\forall l \in \mathrm{L}.\forall(\mathsf{Some}\, x', \_) \in V.x' \neq l$
HF.1 implique trivialement **C**. △
□

Lemme 7.3.3.

Soit	$v \lhd \mathbf{\{}[(\mathsf{Some}\, x', v')] \sqcup V'\mathbf{\}}$
Si	$\langle\mathcal{A}\rangle\Gamma, \{x'\} \uplus L \vdash_{\mathcal{Z}} \mathsf{SEQUENCE}\,\Phi$
alors	$\neg(\langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathsf{SEQUENCE}\,\Phi)$

Preuve du lemme 7.3.3

Soit $v \lhd \mathbf{\{}V\mathbf{\}}$.

Reprenons l'énoncé du lemme 7.3.2 et effectuons-y les renommages suivants :

— $x \xleftarrow{\alpha} x''$

— $\mathrm{L} \xleftarrow{\alpha} \{x'\} \uplus \mathrm{L}$

Il devient alors :

$$\begin{array}{ll} Si & \langle\mathcal{A}\rangle\Gamma, \{x'\} \uplus \mathrm{L} \vdash_{\mathcal{L}} \mathsf{SEQUENCE}\,\Phi \\ et & \langle\mathcal{A}\rangle\Gamma, x \vdash \mathbf{\{}V\mathbf{\}} : \mathsf{SEQUENCE}\,\Phi \\ alors & \forall l \in \{x'\} \uplus \mathrm{L}. \forall(\mathsf{Some}\,x'', \_) \in V. x'' \neq l \end{array}$$

Évaluons partiellement ce lemme en choisissant $l = x'$:

$$\begin{array}{ll} Si & \langle\mathcal{A}\rangle\Gamma, \{x'\} \uplus \mathrm{L} \vdash_{\mathcal{L}} \mathsf{SEQUENCE}\,\Phi \\ et & \langle\mathcal{A}\rangle\Gamma, x \vdash \mathbf{\{}V\mathbf{\}} : \mathsf{SEQUENCE}\,\Phi \\ alors & \forall(\mathsf{Some}\,x'', \_) \in V. x'' \neq x' \end{array}$$

Cet énoncé est équivalent à :

$$\begin{array}{ll} Si & \langle\mathcal{A}\rangle\Gamma, \{x'\} \uplus \mathrm{L} \vdash_{\mathcal{L}} \mathsf{SEQUENCE}\,\Phi \\ et & \langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathsf{SEQUENCE}\,\Phi \\ alors & V \not\lhd [(\mathsf{Some}\,x', v')] \sqcup V' \end{array}$$

Prenons-en finalement une contraposition et nous obtenons notre lemme 7.3.3. □

Lemme 7.3.4.

$$\begin{array}{ll} Si & \langle\mathcal{A}\rangle\Gamma, \{x'\} \uplus L \vdash_{\mathcal{L}} \mathit{CHOICE}\,\mathcal{F} \\ alors & \neg(\langle\mathcal{A}\rangle\Gamma, x \vdash (x' : v') : \mathit{CHOICE}\,\mathcal{F}) \end{array}$$

Preuve du lemme 7.3.4

La démonstration se fera par induction sur $\mathcal{F}$.

— $f' \lhd (l', \tau', \mathrm{T}', \sigma', s')$
$\mathcal{F} \lhd [f'] \sqcup \mathcal{F}'$
et $v \lhd x' : v'$

L'autre hypothèse est

— $\mathbf{H} : \langle\mathcal{A}\rangle\Gamma, \{x'\} \uplus \mathrm{L} \vdash_{\mathcal{L}} \mathsf{CHOICE}\,\mathcal{F}$
Appliquons la règle [3] (page 148) :

$$\frac{\begin{array}{c} f' \lhd (l', \tau', \mathrm{T}', \sigma', s') \\ \langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{L}} \mathrm{T}' \qquad \langle\mathcal{A}\rangle\Gamma, \{l'\} \uplus \mathrm{L} \vdash_{\mathcal{L}} \mathsf{CHOICE}\,\mathcal{F}' \end{array}}{\langle\mathcal{A}\rangle\Gamma, \mathrm{L} \vdash_{\mathcal{L}} \mathsf{CHOICE}\,([f'] \sqcup \mathcal{F}')}\ [3]$$

Ce faisant nous effectuons la substitution :

— $\mathrm{L} \xleftarrow{\beta} \{x'\} \uplus \mathrm{L}$

Étiquetons ses prémisses non redondantes avec les hypothèses de filtre :

— **H.0** : $\langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{L}} \mathrm{T}'$

— **H.1** : $\langle\mathcal{A}\rangle\Gamma, \{l'\} \uplus (\{x'\} \uplus \mathrm{L}) \vdash_{\mathcal{L}} \mathsf{CHOICE}\,\mathcal{F}'$

Prouvons

— **C** : $\neg(\langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathsf{CHOICE}\,\mathcal{F})$

Les seules règle dérivant des jugement sur les types CHOICE sont

$$\frac{f' \lhd (l', \tau', \mathrm{T}', \sigma', s') \qquad \langle\mathcal{A}\rangle\Gamma, "" \vdash v' : \mathrm{T}'}{\langle\mathcal{A}\rangle\Gamma, x \vdash (l' : v') : \mathsf{CHOICE}\,([f'] \sqcup \mathcal{F}')}\ [13]$$

$$\frac{\langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathsf{CHOICE}\,\mathcal{F}'}{\langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathsf{CHOICE}\,([f'] \sqcup \mathcal{F}')}\ [14]$$

Tout d'abord **H.1** implique

— **H.3** : $x' \neq l'$

Par conséquent la règle [13] ne peut s'appliquer. Reprenons l'énoncé du lemme en y effectuant les renommages suivants :

— $\mathcal{F} \xleftarrow{\alpha} \mathcal{F}'$

— $\mathrm{L} \xleftarrow{\alpha} \{l'\} \uplus \mathrm{L}$

Nous obtenons alors :

$$\begin{array}{ll} Si & \langle\mathcal{A}\rangle\Gamma, \{x'\} \uplus (\{l'\} \uplus \mathrm{L}) \vdash_{\mathcal{L}} \mathsf{CHOICE}\,\mathcal{F}' \\ alors & \neg(\langle\mathcal{A}\rangle\Gamma, x \vdash (x' : v') : \mathsf{CHOICE}\,\mathcal{F}') \end{array}$$

Nous reconnaissons en cet énoncé une instance de l'hypothèse d'induction, car sa prémisse est **H.1**. En l'appliquant il vient donc :

— **H.4** : $\neg(\langle\mathcal{A}\rangle\Gamma, x \vdash (x' : v') : \mathsf{CHOICE}\,\mathcal{F}')$

Or, **H.4** est une instance de la négation de la prémisse de la règle [14], qui ne peut donc s'appliquer non plus. Puisque le deux seules règles pouvant dériver $\langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathsf{CHOICE}\,\mathcal{F}$ ne peuvent s'appliquer, cela signifie que **C** est vraie. △

— $\mathcal{F} \lhd [\,]$ et $v \lhd x' : v'$

L'autre hypothèse est

— **H** : $\langle\mathcal{A}\rangle\Gamma, \{x'\} \uplus \mathrm{L} \vdash_{\mathcal{L}} \mathsf{CHOICE}\,\mathcal{F}$

Prouvons

— **C** : $\neg(\langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathsf{CHOICE}\,\mathcal{F})$

Ni la règle [13] (page 146), ni la règle [14] (page 146) ne peuvent s'appliquer car $\mathcal{F} \lhd [\,]$ (aucun filtre n'accepte cette valeur). Puisque le deux seules règles pouvant dériver le jugement $\langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathsf{CHOICE}\,\mathcal{F}$ ne peuvent s'appliquer, cela signifie que **C** est vraie. △

□

Preuve du théorème 7.3.1

Rappelons son énoncé :

> S'il existe une preuve qu'une valeur est d'un type bien labellisé, alors cette preuve est unique.

Les hypothèses :
— **H.1** : $\langle\mathcal{A}\rangle\Gamma \vdash_{\mathcal{L}} \mathrm{T}$
— **H.2** : $\langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathrm{T}$

La preuve se fait par induction sur la forme de T. Nous allons prouver qu'à chaque étape de la dérivation, ou bien aucune règle n'est applicable, ou bien une seule instance d'une seule règle est applicable, grâce à **H.0** et **H.1**.

Tout d'abord on remarque que la seule façon de prouver **H.1** est de prouver :
— **H.1bis** : $\langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{L}} \mathrm{T}$
par la règle [1] (page 148) :

$$\frac{\langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{L}} \mathrm{T}}{\langle\mathcal{A}\rangle\Gamma \vdash_{\mathcal{L}} \mathrm{T}}\ [1]$$

Nous remplaçons alors **H.1** par **H.1bis** dans l'énoncé du théorème. Maintenant, les type à examiner de près sont CHOICE et $\mathsf{SEQUENCE}$.
— $\mathrm{T} \lhd \mathsf{CHOICE}\,([f'] \sqcup \mathcal{F}')$ et $f' \lhd (l', \tau', \mathrm{T}', \sigma', s')$

Les autres hypothèses sont
— **H.1bis** : $\langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{L}} \mathrm{T}$
C'est la règle [3] (page 148) qui s'applique :

$$\frac{\begin{array}{c} f' \lhd (l', \tau', \mathrm{T}', \sigma', s') \\ \langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{L}} \mathrm{T}' \qquad \langle\mathcal{A}\rangle\Gamma, \{l'\} \uplus \mathrm{L} \vdash_{\mathcal{L}} \mathsf{CHOICE}\,\mathcal{F}' \end{array}}{\langle\mathcal{A}\rangle\Gamma, \mathrm{L} \vdash_{\mathcal{L}} \mathsf{CHOICE}\,([f'] \sqcup \mathcal{F}')}\ [3]$$

Appliquons-là directement (i.e. toutes les variables de la règle sont liées aux variables de même nom dans l'environnement du cas courant de la preuve) et étiquetons ses prémisses non redondantes avec les hypothèses de filtre :
— **H.1bis.0** : $\langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{L}} \mathrm{T}'$
— **H.1bis.1** : $\langle\mathcal{A}\rangle\Gamma, \{l'\} \uplus \mathrm{L} \vdash_{\mathcal{L}} \mathsf{CHOICE}\,\mathcal{F}'$

— **H.2** : $\langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathrm{T}$
La règle [13] (page 146) :

$$\frac{f' \lhd (l', \tau', \mathrm{T}', \sigma', s') \qquad \langle\mathcal{A}\rangle\Gamma, \text{""} \vdash v' : \mathrm{T}'}{\langle\mathcal{A}\rangle\Gamma, x \vdash (l' : v') : \mathsf{CHOICE}\,([f'] \sqcup \mathcal{F}')}\ [13]$$

ou [14] (page 146) :

$$\frac{\langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathsf{CHOICE}\,\mathcal{F}'}{\langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathsf{CHOICE}\,([f'] \sqcup \mathcal{F}')}\ [14]$$

peuvent s'appliquer. Nous remarquons d'abord que si $v \not\lhd x' : v'$ alors la seule règle éventuellement applicable serait [14], ce qui démontrerait le théorème. Supposons donc maintenant :
— **HF.0** : $v \lhd x' : v'$
Nous remarquons alors que si $x' \neq l'$, alors la seule règle éventuellement applicable serait [14], ce qui démontrerait le théorème. Supposons donc maintenant :
— **HF.1** : $x' = l'$
Rien ne s'oppose *a priori* à ce que la règle [13] s'applique. *Il nous suffit alors de prouver que la règle* [14] *ne peut s'appliquer*. Pour ce faire, prouvons la négation de sa prémisse, soit à prouver :
— **C** : $\neg(\langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathsf{CHOICE}\,\mathcal{F}')$
HF.1 et **H.1bis.1** impliquent
— **H.3** : $\langle\mathcal{A}\rangle\Gamma, \{x'\} \uplus \mathrm{L} \vdash_{\mathcal{L}} \mathsf{CHOICE}\,\mathcal{F}'$
Rappelons l'énoncé du lemme 7.3.4 :

$$\begin{array}{ll} Si & \langle\mathcal{A}\rangle\Gamma, \{x'\} \uplus \mathrm{L} \vdash_{\mathcal{L}} \mathsf{CHOICE}\,\mathcal{F} \\ alors & \neg(\langle\mathcal{A}\rangle\Gamma, x \vdash (x' : v') : \mathsf{CHOICE}\,\mathcal{F}) \end{array}$$

Appliquons ce lemme à **H.3**. Ce faisant nous effectuons la substitution
— $\mathcal{F} \xleftarrow{\beta} \mathcal{F}'$
Nous obtenons alors :
— **H.4** : $\neg(\langle\mathcal{A}\rangle\Gamma, x \vdash (x' : v') : \mathsf{CHOICE}\,\mathcal{F}')$
H.4 et **HF.0** impliquent **C**. $\triangle$

— $\mathrm{T} \lhd \mathsf{SEQUENCE}\,([\varphi'] \sqcup \Phi')$ et $\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s')$ Les autres hypothèses sont
— **H.1bis** : $\langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{L}} \mathrm{T}$

La règle [5] (page 149) s'applique :

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \\ \langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{L}} \mathrm{T}' \qquad \langle\mathcal{A}\rangle\Gamma, \{l'\} \uplus \mathrm{L} \vdash_{\mathcal{L}} \mathsf{SEQUENCE}\,\Phi'\end{array}}{\langle\mathcal{A}\rangle\Gamma, \mathrm{L} \vdash_{\mathcal{L}} \mathsf{SEQUENCE}\,([\varphi'] \sqcup \Phi')}\ [5]$$

Appliquons-là directement et étiquetons ses prémisses non redondantes avec les hypothèses de filtre :
— **H.1bis.0** : $\langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{L}} \mathrm{T}'$
— **H.1bis.1** : $\langle\mathcal{A}\rangle\Gamma, \{l'\} \uplus \mathrm{L} \vdash_{\mathcal{L}} \mathsf{SEQUENCE}\ \Phi'$
— **H.2** : $\langle\mathcal{A}\rangle\Gamma \vdash v : \mathrm{T}$. La forme de T implique que peuvent s'appliquer les règles [15] (page 147) et [16] (page 147) :

$$\frac{\begin{array}{cc}\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') & V \lhd [(\mathsf{Some}\, l', v')] \sqcup V' \\ \langle\mathcal{A}\rangle\Gamma, "" \vdash v' : \mathrm{T}' & \langle\mathcal{A}\rangle\Gamma, x \vdash \mathbf{\{}V'\mathbf{\}} : \mathsf{SEQUENCE}\ \Phi'\end{array}}{\langle\mathcal{A}\rangle\Gamma, x \vdash \mathbf{\{}V\mathbf{\}} : \mathsf{SEQUENCE}\,([\varphi'] \sqcup \Phi')}\ [15]$$

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \\ s' \lhd \mathsf{Some}\,\mathsf{OPTIONAL} \qquad \langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathsf{SEQUENCE}\ \Phi'\end{array}}{\langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathsf{SEQUENCE}\,([\varphi'] \sqcup \Phi')}\ [16]$$

Remarquons d'abord que si $v \not\lhd \mathbf{\{}V\mathbf{\}}$, alors la seule règle éventuellement applicable serait [16], ce qui démontrerait le théorème. Supposons donc :
— **H.3** : $v \lhd \mathbf{\{}V\mathbf{\}}$
Remarquons de plus que si $V \not\lhd [(\mathsf{Some}\, l', v')] \sqcup V'$, alors la seule règle éventuellement applicable serait [16], ce qui démontrerait le théorème. Supposons donc :
— **H.4** : $V \lhd [(\mathsf{Some}\, l', v')] \sqcup V'$
Remarquons ensuite que si $s' \not\lhd \mathsf{Some}\,\mathsf{OPTIONAL}$, la seule règle éventuellement applicable serait [15], ce qui démontrerait le théorème. Supposons donc :
— **H.5** : $s' \lhd \mathsf{Some}\,\mathsf{OPTIONAL}$
Maintenant, les règles [15] et [16] peuvent *a priori* être appliquées, mais *nous allons montrer que seule la règle* [15] *est éventuellement applicable.* Appliquons directement la règle [15] et étiquetons ses prémisses non redondantes avec les hypothèses précédentes :
— **H.2.0** : $\langle\mathcal{A}\rangle\Gamma, "" \vdash v' : \mathrm{T}'$
— **H.2.1** : $\langle\mathcal{A}\rangle\Gamma, x \vdash \mathbf{\{}V'\mathbf{\}} : \mathsf{SEQUENCE}\ \Phi'$
Nous devons considérer deux aspects :

1. Nous devons prouver que **H.0** et **H.1bis** permettent de conclure que
   — **C.0** : $\forall(\mathsf{Some}\, x', \_) \in V'.x' \neq l'$
   En effet, ainsi **H.4**, qui est une prémisse de **H.2**, serait l'unique projection possible. En d'autres termes :
   — **H.4** $\Rightarrow$ **C.0**

(d'où l'unicité de la projection). Reprenons l'énoncé du lemme 7.3.2 page 149 et effectuons-y les renommages suivants :
— $\mathrm{L} \xleftarrow{\alpha} \{l'\} \uplus \mathrm{L}$
— $V \xleftarrow{\alpha} V'$
— $\Phi \xleftarrow{\alpha} \Phi'$
Nous obtenons alors le lemme suivant :

$$\begin{array}{ll} \mathit{Si} & \langle\mathcal{A}\rangle\Gamma, \{l'\} \uplus \mathrm{L} \vdash_{\mathcal{L}} \mathsf{SEQUENCE}\,\Phi' \\ \mathit{et} & \langle\mathcal{A}\rangle\Gamma, x \vdash \mathbf{\{}V'\mathbf{\}} : \mathsf{SEQUENCE}\,\Phi' \\ \mathit{alors} & \forall l \in \{l'\} \uplus \mathrm{L}.\forall(\mathsf{Some}\,x', \_) \in V'.x' \neq l \end{array}$$

Évaluons partiellement ce lemme en prenant $l = l'$. Il devient alors :

$$\begin{array}{ll} \mathit{Si} & \langle\mathcal{A}\rangle\Gamma, \{l'\} \uplus \mathrm{L} \vdash_{\mathcal{L}} \mathsf{SEQUENCE}\,\Phi' \\ \mathit{et} & \langle\mathcal{A}\rangle\Gamma, x \vdash \mathbf{\{}V'\mathbf{\}} : \mathsf{SEQUENCE}\,\Phi' \\ \mathit{alors} & \forall(\mathsf{Some}\,x', \_) \in V'.x' \neq l' \end{array}$$

Appliquons ce dernier à **H.1bis.1** et **H.2.1**. Il vient **C.0**. ◊

2. *Nous devons prouver que la règle* [16] *(page 147) ne peut s'appliquer.* Prouvons la négation de sa prémisse, soit à prouver :
— **C.1** : $\neg(\langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathsf{SEQUENCE}\,\Phi')$
Reprenons l'énoncé du lemme 7.3.3 et effectuons-y les renommages suivants :
— $x' \xleftarrow{\alpha} l'$
— $\Phi \xleftarrow{\alpha} \Phi'$
Nous obtenons alors le lemme équivalent :

$$\begin{array}{ll} \mathit{Soit} & v \lhd \mathbf{\{}[(\mathsf{Some}\,l', v')] \sqcup V'\mathbf{\}} \\ \mathit{Si} & \langle\mathcal{A}\rangle\Gamma, \{l'\} \uplus \mathrm{L} \vdash_{\mathcal{L}} \mathsf{SEQUENCE}\,\Phi' \\ \mathit{alors} & \neg(\langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathsf{SEQUENCE}\,\Phi') \end{array}$$

En appliquant ce lemme respectivement à **H.4** et **H.1bis.1** il vient **C.1**. ◊ △

□

Chapitre 8

Sémantique

8.1 Codage

```
type primitive_val =
  P_int of int
| P_string of string
| P_num_str of string
| P_false
| P_true
| P_null
| P_plus_inf
| P_min_inf
| P_zero_real

type code = tag_id × contents

and contents =
  Primitive of primitive_val
| Const of code list
```

$$\frac{\langle\mathcal{A}\rangle\Gamma \vdash v : \tau'\,\mathrm{T} \Rightarrow \tilde{v}'}{\langle\mathcal{A}\rangle\Gamma \vdash v : ((\mathsf{Tag}\ \psi\ \mathsf{EXPLICIT}) :: \tau')\,\mathrm{T} \Rightarrow (\psi, \mathsf{Const}\,[\tilde{v}'])}\ [1]$$

$$\frac{x' \in \mathcal{A} \qquad \Gamma(x') \lhd (\alpha', \tau', \{(\mathrm{T}', \sigma')\}) \qquad \langle\mathcal{A}\rangle\Gamma \vdash v : \tau'\,\mathrm{T}' \Rightarrow \tilde{v}}{\langle\mathcal{A}\rangle\Gamma \vdash v : [\,]\ (\mathsf{TRef}\,x') \Rightarrow \tilde{v}}\ [2]$$

$$\langle\mathcal{A}\rangle\Gamma \vdash \mathsf{Int}\,(n) : [(\mathsf{Tag}\ \psi\ \_)]\,\mathrm{T} \Rightarrow (\psi, \mathsf{Primitive}\,(\mathsf{P\_int}\,n))\ [3]$$

$$\langle\mathcal{A}\rangle\Gamma \vdash \mathsf{Str}\,(s) : [(\mathsf{Tag}\ \psi\ \_)]\,\mathrm{T} \Rightarrow (\psi, \mathsf{Primitive}\,(\mathsf{P\_string}\,s))\ [4]$$

$$\langle\mathcal{A}\rangle\Gamma \vdash \mathsf{BitStr}\,(s) : [(\mathsf{Tag}\ \psi\ \_)]\,\mathrm{T} \Rightarrow (\psi, \mathsf{Primitive}\,(\mathsf{P\_num\_str}\,s))\ [5]$$

$$\langle\mathcal{A}\rangle\Gamma \vdash \mathsf{TRUE} : [(\mathsf{Tag}\ \psi\ \_)]\,\mathrm{T} \Rightarrow (\psi, \mathsf{Primitive}\,\mathsf{P\_true})\ [6]$$

$$\langle\mathcal{A}\rangle\Gamma \vdash \mathsf{FALSE} : [(\mathsf{Tag}\ \psi\ \_)]\,\mathrm{T} \Rightarrow (\psi, \mathsf{Primitive}\,\mathsf{P\_false})\ [7]$$

$$\langle\mathcal{A}\rangle\Gamma \vdash \mathsf{NULL} : [(\mathsf{Tag}\ \psi\ \_)]\,\mathrm{T} \Rightarrow (\psi, \mathsf{Primitive}\,\mathsf{P\_null})\ [8]$$

$$\langle\mathcal{A}\rangle\Gamma \vdash \mathsf{PLUS\text{-}INFINITY} : [(\mathsf{Tag}\ \psi\ \_)]\,\mathrm{T} \Rightarrow (\psi, \mathsf{Primitive}\,\mathsf{P\_plus\_inf})\ [9]$$

$$\langle\mathcal{A}\rangle\Gamma \vdash \mathsf{MINUS\text{-}INFINITY} : [(\mathsf{Tag}\ \psi\ \_)]\,\mathrm{T} \Rightarrow (\psi, \mathsf{Primitive}\,\mathsf{P\_min\_inf})\ [10]$$

$$\langle\mathcal{A}\rangle\Gamma \vdash \mathsf{0.0} : [(\mathsf{Tag}\ \psi\ \_)]\,\mathrm{T} \Rightarrow (\psi, \mathsf{Primitive}\,\mathsf{P\_zero\_real})\ [11]$$

$$\frac{\mathrm{T} \lhd \mathsf{ENUMERATED}\,([(y, \mathsf{Int}\,n)] \sqcup \mathcal{C}')}{\langle\mathcal{A}\rangle\Gamma \vdash \mathsf{Enum}\,(y) : [(\mathsf{Tag}\ \psi\ \_)]\,\mathrm{T} \Rightarrow (\psi, \mathsf{Primitive}\,(\mathsf{P\_int}\,n))}\ [12]$$

$$\frac{\begin{array}{c}\mathrm{T} \lhd \mathsf{SEQUENCE\ OF}\,\tau'\,\mathrm{T}'\,\sigma' \qquad \langle\mathcal{A}\rangle\Gamma \vdash v' : \tau'\,\mathrm{T}' \Rightarrow \tilde{v}' \\ \langle\mathcal{A}\rangle\Gamma \vdash \{V'\} : \tau\,\mathrm{T} \Rightarrow (\psi, \mathsf{Const}\,\tilde{V}') \qquad \tilde{v} \triangleq (\psi, \mathsf{Const}\,(\tilde{v}' :: \tilde{V}'))\end{array}}{\langle\mathcal{A}\rangle\Gamma \vdash \{(\mathsf{None}, v') :: V'\} : \tau\,\mathrm{T} \Rightarrow \tilde{v}}\ [13]$$

$$\frac{\begin{array}{c}\mathrm{T} \lhd \mathsf{SET\ OF}\,\tau'\,\mathrm{T}'\,\sigma' \qquad \langle\mathcal{A}\rangle\Gamma \vdash v' : \tau'\,\mathrm{T}' \Rightarrow \tilde{v}' \\ \langle\mathcal{A}\rangle\Gamma \vdash \{V'\} : \tau\,\mathrm{T} \Rightarrow (\psi, \mathsf{Const}\,\tilde{V}') \qquad \tilde{v} \triangleq (\psi, \mathsf{Const}\,([\tilde{v}'] \sqcup \tilde{V}'))\end{array}}{\langle\mathcal{A}\rangle\Gamma \vdash \{[(\mathsf{None}, v')] \sqcup V'\} : \tau\,\mathrm{T} \Rightarrow \tilde{v}}\ [14]$$

$$\frac{\mathrm{T} \lhd (\mathsf{SET\ OF} \mid \mathsf{SEQUENCE\ OF})\,\tau'\,\mathrm{T}'\,\sigma'}{\langle\mathcal{A}\rangle\Gamma \vdash \{[\,]\} : [(\mathsf{Tag}\ \psi\ \_)]\,\mathrm{T} \Rightarrow (\psi, \mathsf{Const}\,[\,])}\ [15]$$

$$\frac{f' \lhd (l', \tau', \mathrm{T}', \sigma', s') \qquad \langle\mathcal{A}\rangle\Gamma \vdash v' : \tau'\,\mathrm{T}' \Rightarrow \tilde{v}}{\langle\mathcal{A}\rangle\Gamma \vdash (l' : v') : \tau\,(\mathsf{CHOICE}\,([f'] \sqcup \mathcal{F}')) \Rightarrow \tilde{v}}\ [16]$$

$$\frac{\langle\mathcal{A}\rangle\Gamma \vdash v : \tau\,(\mathsf{CHOICE}\,\mathcal{F}') \Rightarrow \tilde{v}}{\langle\mathcal{A}\rangle\Gamma \vdash v : \tau\,(\mathsf{CHOICE}\,([f'] \sqcup \mathcal{F}')) \Rightarrow \tilde{v}}\ [17]$$

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \\ V \lhd [(\mathsf{Some}\,l', v')] \sqcup V' \qquad \langle\mathcal{A}\rangle\Gamma \vdash v' : \tau'\,\mathrm{T}' \Rightarrow \tilde{v}' \\ \langle\mathcal{A}\rangle\Gamma \vdash \{V'\} : \tau\,(\mathsf{SEQUENCE}\,\Phi') \Rightarrow (\psi, \mathsf{Const}\,\tilde{V}') \\ \tilde{v} \triangleq (\psi, \mathsf{Const}\,(\tilde{v}' :: \tilde{V}'))\end{array}}{\langle\mathcal{A}\rangle\Gamma \vdash \{V\} : \tau\,(\mathsf{SEQUENCE}\,(\varphi' :: \Phi')) \Rightarrow \tilde{v}}\ [18]$$

$$\frac{\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \qquad s' \lhd \mathsf{Some}\,\mathsf{OPTIONAL} \qquad \langle\mathcal{A}\rangle\Gamma \vdash v : \tau\,(\mathsf{SEQUENCE}\,\Phi') \Rightarrow \tilde{v}}{\langle\mathcal{A}\rangle\Gamma \vdash v : \tau\,(\mathsf{SEQUENCE}\,([\varphi'] \sqcup \Phi')) \Rightarrow \tilde{v}}\ [19]$$

$$\frac{\mathrm{T} \lhd \mathsf{SEQUENCE}\,[\,]}{\langle\mathcal{A}\rangle\Gamma \vdash \mathbf{\{[\,]\}} : [(\mathsf{Tag}\ \psi\ \_)]\,\mathrm{T} \Rightarrow (\psi, \mathsf{Const}\,[\,])}\ [20]$$

$$\frac{\langle\mathcal{A}\rangle\Gamma \vdash v : \tau\,(\mathsf{SEQUENCE}\,\Phi) \Rightarrow (\psi, \mathsf{Const}\,\tilde{V}) \qquad \pi\ \textit{est une permutation sur}\ \tilde{V}}{\langle\mathcal{A}\rangle\Gamma \vdash v : \tau\,(\mathsf{SET}\,\Phi) \Rightarrow (\psi, \mathsf{Const}\,(\pi\ (\tilde{V})))}\ [21]$$

8.2 Contrôle sémantique des types

8.2.1 Inclusion d'étiquetages

$$\frac{\langle\mathcal{A}\rangle\Gamma, [\,] \vDash_{95} e_0 \sqsubseteq e_1}{\langle\mathcal{A}\rangle\Gamma \vdash e_0 \sqsubseteq e_1}\ [0] \qquad \frac{\mathrm{H} \lhd [(e_0, e_1)] \sqcup \mathrm{H}'}{\langle\mathcal{A}\rangle\Gamma, \mathrm{H} \vDash_{95} e_0 \sqsubseteq e_1}\ [1]$$

$$\frac{x' \in \mathcal{A} \qquad \Gamma(x') \lhd (\alpha', \tau', \{(\mathrm{T}', \sigma')\}) \qquad \langle\mathcal{A}\rangle\Gamma, \mathrm{H} \vDash_{95} (\tau', \mathrm{T}') \sqsubseteq e_1}{\langle\mathcal{A}\rangle\Gamma, \mathrm{H} \vDash_{95} ([\,], \mathsf{TRef}\,x') \sqsubseteq e_1}\ [2]$$

$$\frac{x' \in \mathcal{A} \qquad \Gamma(x') \lhd (\alpha', \tau', \{(\mathrm{T}', \sigma')\}) \qquad \langle\mathcal{A}\rangle\Gamma, \mathrm{H} \vDash_{95} e_0 \sqsubseteq (\tau', \mathrm{T}')}{\langle\mathcal{A}\rangle\Gamma, \mathrm{H} \vDash_{95} e_0 \sqsubseteq ([\,], \mathsf{TRef}\,x')}\ [3]$$

$$\frac{\langle\mathcal{A}\rangle\Gamma, \mathrm{H} \vDash_{95} (\tau'_0, \mathrm{T}_0) \sqsubseteq (\tau'_1, \mathrm{T}_1)}{\langle\mathcal{A}\rangle\Gamma, \mathrm{H} \vDash_{95} (t :: \tau'_0, \mathrm{T}_0) \sqsubseteq (t :: \tau'_1, \mathrm{T}_1)}\ [4]$$

$$\frac{\mathrm{T} \lhd \mathsf{CHOICE}\,([f'] \sqcup \mathcal{F}') \qquad f' \lhd (l', \tau', \mathrm{T}', \sigma', s') \qquad \langle\mathcal{A}\rangle\Gamma, (e_0, e_1) :: \mathrm{H} \vDash_{95} (\tau', \mathrm{T}') \sqsubseteq e_1 \qquad \langle\mathcal{A}\rangle\Gamma, \mathrm{H} \vDash_{95} ([\,], \mathsf{CHOICE}\,\mathcal{F}') \sqsubseteq e_1}{\langle\mathcal{A}\rangle\Gamma, \mathrm{H} \vDash_{95} ([\,], \mathrm{T})\,\mathbf{as}\,e_0 \sqsubseteq e_1}\ [5]$$

$$\langle\mathcal{A}\rangle\Gamma, \mathrm{H} \vDash_{95} ([\,], \mathsf{CHOICE}\,[\,]) \sqsubseteq e_1\ [6]$$

$$\frac{\mathrm{T} \lhd \mathsf{CHOICE}\,([f'] \sqcup \mathcal{F}') \qquad f' \lhd (l', \tau', \mathrm{T}', \sigma', s') \qquad \langle\mathcal{A}\rangle\Gamma, (e_0, e_1) :: \mathrm{H} \vDash_{95} e_0 \sqsubseteq (\tau', \mathrm{T}')}{\langle\mathcal{A}\rangle\Gamma, \mathrm{H} \vDash_{95} e_0 \sqsubseteq ([\,], \mathrm{T})\,\mathbf{as}\,e_1}\ [7]$$

$$\frac{\mathrm{T}_0 \not\lhd \mathsf{CHOICE}\ \ \_ \mid \mathsf{TRef}\ \ \_ \mid \_ < \_\_ \qquad \mathrm{T}_1 \not\lhd \mathsf{CHOICE}\ \ \_ \mid \mathsf{TRef}\ \ \_ \mid \_ < \_\_}{\langle\mathcal{A}\rangle\Gamma, \mathrm{H} \vDash_{95} ([\,], \mathrm{T}_0) \sqsubseteq ([\,], \mathrm{T}_1)}\ [8]$$

Propriété 8.2.1.1 (Réflexivité)**.**
$\forall a : \mathcal{E} \times \mathcal{T}.\langle\mathcal{A}\rangle\Gamma \vdash a \sqsubseteq a$

Preuve de la propriété 8.2.1.1. Laissée au lecteur.

8.2.2 Étiquetages de types non disjoints

Définition 8.2.2.1.

Soient deux étiquetages e_0 *et* e_1*.*
Ils sont dits non disjoints *dans un environnement de types* $\langle\mathcal{A}\rangle\Gamma$ *si*
$\exists x : \mathcal{E} \times \mathcal{T}.(\langle\mathcal{A}\rangle\Gamma \vdash x \sqsubseteq e_0 \wedge \langle\mathcal{A}\rangle\Gamma \vdash x \sqsubseteq e_1)$
On notera : $\langle\mathcal{A}\rangle\Gamma \vdash e_0 \parallel e_1$

Propriété 8.2.2.2 (Réflexivité)**.**

$$\forall a : \mathcal{E} \times \mathcal{T}.\langle\mathcal{A}\rangle\Gamma \vdash a \parallel a$$

Preuve de la propriété 8.2.2.2

D'après 8.2.1.1 : $\forall a : \mathcal{E} \times \mathcal{T}.\langle\mathcal{A}\rangle\Gamma \vdash a \sqsubseteq a$. D'où :

$$\forall a : \mathcal{E} \times \mathcal{T}.\langle\mathcal{A}\rangle\Gamma \vdash a \sqsubseteq a \Rightarrow \exists x : \mathcal{E} \times \mathcal{T}.\langle\mathcal{A}\rangle\Gamma \vdash x \sqsubseteq a$$
$$\Leftrightarrow \exists x : \mathcal{E} \times \mathcal{T}.(\langle\mathcal{A}\rangle\Gamma \vdash x \sqsubseteq a \wedge \langle\mathcal{A}\rangle\Gamma \vdash x \sqsubseteq a) \Leftrightarrow \langle\mathcal{A}\rangle\Gamma \vdash a \parallel a$$

□

Propriété 8.2.2.3 (Symétrie)**.**

$$\forall a, b : \mathcal{E} \times \mathcal{T}.(\langle\mathcal{A}\rangle\Gamma \vdash a \parallel b \Rightarrow \langle\mathcal{A}\rangle\Gamma \vdash b \parallel a)$$

Preuve de la propriété 8.2.2.3

Par définition :

$$\begin{aligned}
\langle\mathcal{A}\rangle\Gamma \vdash a \parallel b &\Leftrightarrow \exists x : \mathcal{E} \times \mathcal{T}.(\langle\mathcal{A}\rangle\Gamma \vdash x \sqsubseteq a \wedge \langle\mathcal{A}\rangle\Gamma \vdash x \sqsubseteq b) \\
&\Leftrightarrow \exists x : \mathcal{E} \times \mathcal{T}.(\langle\mathcal{A}\rangle\Gamma \vdash x \sqsubseteq b \wedge \langle\mathcal{A}\rangle\Gamma \vdash x \sqsubseteq a) \\
&\Leftrightarrow \langle\mathcal{A}\rangle\Gamma \vdash b \parallel a. \qquad \square
\end{aligned}$$

Théorème 8.2.2.4.

Soient $\langle\mathcal{A}\rangle\Gamma$ *un environnement de types*
et a, b *et* c *trois étiquetages.*
Alors : $\langle\mathcal{A}\rangle\Gamma \vdash a \sqsubseteq b \wedge \neg(\langle\mathcal{A}\rangle\Gamma \vdash c \parallel b) \Rightarrow \neg(\langle\mathcal{A}\rangle\Gamma \vdash a \sqsubseteq c)$

Preuve du théorème 8.2.2.4

Par définition : $\langle\mathcal{A}\rangle\Gamma \vdash c \parallel b \Leftrightarrow \exists x.(\langle\mathcal{A}\rangle\Gamma \vdash x \sqsubseteq c \wedge \langle\mathcal{A}\rangle\Gamma \vdash x \sqsubseteq b)$
Alors : $\neg(\langle\mathcal{A}\rangle\Gamma \vdash c \parallel b) \Leftrightarrow \forall x.(\neg(\langle\mathcal{A}\rangle\Gamma \vdash x \sqsubseteq c) \vee \neg(\langle\mathcal{A}\rangle\Gamma \vdash x \sqsubseteq b))$
Prenons $x = a$. On a : $\neg(\langle\mathcal{A}\rangle\Gamma \vdash a \sqsubseteq c) \vee \neg(\langle\mathcal{A}\rangle\Gamma \vdash a \sqsubseteq b)$
Puisque $\langle\mathcal{A}\rangle\Gamma \vdash a \sqsubseteq b$, il reste : $\neg(\langle\mathcal{A}\rangle\Gamma \vdash a \sqsubseteq c)$. □

Théorème 8.2.2.5.

Soient	$\langle\mathcal{A}\rangle\Gamma$ *un environnement de types*
et	e *un étiquetage*
et	E *un ensemble d'étiquetages.*
Si	$\forall x, y \in E.(\langle\mathcal{A}\rangle\Gamma \vdash x \parallel y \Rightarrow x = y)$
et	$\forall z \in E.\neg(\langle\mathcal{A}\rangle\Gamma \vdash z \parallel e)$
alors	$\forall x, y \in \{e\} \cup E.(\langle\mathcal{A}\rangle\Gamma \vdash x \parallel y \Rightarrow x = y)$

Preuve du théorème 8.2.2.5

La propriété de réflexivité 8.2.2.2 et $\forall z \in \mathrm{E}.\neg(\langle\mathcal{A}\rangle\Gamma \vdash z \parallel e)$ impliquent $e \notin \mathrm{E}$. Donc $\forall x, y \in e \cup \mathrm{E}.(\langle\mathcal{A}\rangle\Gamma \vdash x \parallel y \Rightarrow x = y)$ est équivalente à la conjonction des cas suivants :

$$\begin{cases} x = e \wedge \forall y \in \mathrm{E}.(\langle\mathcal{A}\rangle\Gamma \vdash x \parallel y \Rightarrow x = y) \\ \forall x, y \in \mathrm{E}.(\langle\mathcal{A}\rangle\Gamma \vdash x \parallel y \Rightarrow x = y) \\ y = e \wedge \forall x \in \mathrm{E}.(\langle\mathcal{A}\rangle\Gamma \vdash x \parallel y \Rightarrow x = y) \\ x = e \wedge y = e \wedge (\langle\mathcal{A}\rangle\Gamma \vdash x \parallel y \Rightarrow x = y) \end{cases}$$

Le second cas est la première des deux hypothèses. Le troisième se ramène au premier en appliquant la propriété de symétrie 8.2.2.3 et en renommant x en y. Le dernier cas est trivial. Le premier cas est vrai à cause de la seconde hypothèse $\forall z \in \mathrm{E}.\neg(\langle\mathcal{A}\rangle\Gamma \vdash z \parallel e)$ qui implique, avec la propriété de symétrie : $\neg(\langle\mathcal{A}\rangle\Gamma \vdash e \parallel y)$. □

8.2.3 Contrôle sémantique des types

let is_built_in_tag = **function**
(UNIVERSAL,
Imm (1 | 2 | 3 | 4 | 5 | 6 | 10 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 25 | 26 | 27)) → **true**
| _ → **false**

On définit une fonction $\mathcal{D}$ comme étant la plus petite relation définie par le système suivant :

$$\frac{\mathrm{T} \neq \mathsf{TRef}\ \_ \mid \_ < \_ \_ \mid \mathsf{CHOICE}\ \_}{\vdash_{\mathcal{D}} (\psi, \mathsf{Primitive}\_) \to ([\mathsf{Tag}\ \psi\ \mathsf{IMPLICIT}], \mathrm{T})}\ [1]$$

$$\frac{\mathrm{is\_built\_in\_tag}\,(\psi) \qquad \mathrm{T} \neq \mathsf{TRef}\ \_ \mid \_ < \_ \_ \mid \mathsf{CHOICE}\ \_}{\vdash_{\mathcal{D}} (\psi, \mathsf{Const}\_) \to ([\mathsf{Tag}\ \psi\ \mathsf{IMPLICIT}], \mathrm{T})}\ [2]$$

$$\frac{\vdash_{\mathcal{D}} \overline{v}' \to (\tau', \mathrm{T})}{\vdash_{\mathcal{D}} (\psi, \mathsf{Const}\,[\overline{v}']) \to ((\mathsf{Tag}\ \psi\ \mathsf{EXPLICIT}) :: \tau', \mathrm{T})}\ [3]$$

$$\frac{\langle\mathcal{A}\rangle\Gamma \models \tilde{v}' : \tau'\,\mathrm{T}}{\langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Const}\,[\tilde{v}']) : ((\mathsf{Tag}\ \psi\ \mathsf{EXPLICIT}) :: \tau')\,\mathrm{T}}\ [1]$$

$$\frac{x' \in \mathcal{A} \qquad \Gamma(x') \lhd (\alpha', \tau', \{(\mathrm{T}', \sigma')\}) \qquad \langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau'\,\mathrm{T}'}{\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : [\,]\ (\mathsf{TRef}\,x')}\ [2]$$

$$\frac{\psi \lhd (\mathsf{UNIVERSAL}, \mathsf{Imm}\,1)}{\langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Primitive}\,(\mathsf{P\_true} \mid \mathsf{P\_false})) : [\mathsf{Tag}\ \psi\ \mathsf{IMPLICIT}]\,\mathrm{T}}\ [3]$$

$$\frac{\psi \lhd (\mathsf{UNIVERSAL}, \mathsf{Imm}\,2)}{\langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Primitive}\,(\mathsf{P\_int}\ \_)) : [\mathsf{Tag}\ \psi\ \mathsf{IMPLICIT}]\,\mathrm{T}}\ [4]$$

$$\frac{\psi \lhd (\mathsf{UNIVERSAL}, \mathsf{Imm}\,(3 \mid 4))}{\langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Primitive}\,(\mathsf{P\_num\_str}\ \_)) : [\mathsf{Tag}\ \psi\ \mathsf{IMPLICIT}]\ \mathrm{T}}\ [5]$$

$$\frac{\psi \lhd (\mathsf{UNIVERSAL}, \mathsf{Imm}\,5)}{\langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Primitive}\,\mathsf{P\_null}) : [\mathsf{Tag}\ \psi\ \mathsf{IMPLICIT}]\ \mathrm{T}}\ [6]$$

$$\frac{\psi \lhd (\mathsf{UNIVERSAL}, \mathsf{Imm}\,(18 \mid 19 \mid 20 \mid 21 \mid 22 \mid 25 \mid 26 \mid 27))}{\langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Primitive}\,(\mathsf{P\_string}\ \_)) : [\mathsf{Tag}\ \psi\ \mathsf{IMPLICIT}]\ \mathrm{T}}\ [7]$$

$$\frac{\mathrm{T} \lhd \mathsf{ENUMERATED}\ ([(\_, \mathsf{Int}\, n)] \sqcup \mathcal{C}') \qquad \psi \lhd (\mathsf{UNIVERSAL}, \mathsf{Imm}\, 10)}{\langle \mathcal{A} \rangle \Gamma \models (\psi, \mathsf{Primitive}\, (\mathsf{P\_int}\, n)) : [\mathsf{Tag}\ \psi\ \mathsf{IMPLICIT}]\ \mathrm{T}}\ [8]$$

$$\frac{\psi \lhd (\mathsf{UNIVERSAL}, \mathsf{Imm}\, 9)}{\langle \mathcal{A} \rangle \Gamma \models (\psi, \mathsf{Primitive}\, \mathsf{P\_plus\_inf}) : [\mathsf{Tag}\ \psi\ \mathsf{IMPLICIT}]\ \mathrm{T}}\ [9]$$

$$\frac{\psi \lhd (\mathsf{UNIVERSAL}, \mathsf{Imm}\, 9)}{\langle \mathcal{A} \rangle \Gamma \models (\psi, \mathsf{Primitive}\, \mathsf{P\_min\_inf}) : [\mathsf{Tag}\ \psi\ \mathsf{IMPLICIT}]\ \mathrm{T}}\ [10]$$

$$\frac{\psi \lhd (\mathsf{UNIVERSAL}, \mathsf{Imm}\, 9)}{\langle \mathcal{A} \rangle \Gamma \models (\psi, \mathsf{Primitive}\, \mathsf{P\_zero\_real}) : [\mathsf{Tag}\ \psi\ \mathsf{IMPLICIT}]\ \mathrm{T}}\ [11]$$

$$\frac{\begin{array}{c}\mathrm{T} \lhd (\mathsf{SET\ OF} \mid \mathsf{SEQUENCE\ OF})\, \tau'\, \mathrm{T}'\, \sigma' \\ \langle \mathcal{A} \rangle \Gamma \models \tilde{v}' : \tau'\, \mathrm{T}' \qquad \langle \mathcal{A} \rangle \Gamma \models (\psi, \mathsf{Const}\, \tilde{V}') : \tau\, \mathrm{T}\end{array}}{\langle \mathcal{A} \rangle \Gamma \models (\psi, \mathsf{Const}\, ([\tilde{v}'] \sqcup \tilde{V}')) : \tau\, \mathrm{T}}\ [12]$$

$$\frac{\psi \lhd (\mathsf{UNIVERSAL}, \mathsf{Imm}\, (16 \mid 17))}{\langle \mathcal{A} \rangle \Gamma \models (\psi, \mathsf{Const}\, [\,]) : [\mathsf{Tag}\ \psi\ \mathsf{IMPLICIT}]\ \mathrm{T}}\ [13]$$

$$\frac{f' \lhd (l', \tau', \mathrm{T}', \sigma', s') \qquad \langle \mathcal{A} \rangle \Gamma \vdash \mathcal{D}(\tilde{v}) \sqsubseteq (\tau', \mathrm{T}') \qquad \langle \mathcal{A} \rangle \Gamma \models \tilde{v} : \tau'\, \mathrm{T}'}{\langle \mathcal{A} \rangle \Gamma \models \tilde{v} : [\,]\ (\mathsf{CHOICE}\, ([f'] \sqcup \mathcal{F}'))}\ [14]$$

$$\frac{\langle \mathcal{A} \rangle \Gamma \models \tilde{v} : \tau\ (\mathsf{CHOICE}\, \mathcal{F}')}{\langle \mathcal{A} \rangle \Gamma \models \tilde{v} : \tau\ (\mathsf{CHOICE}\, ([f'] \sqcup \mathcal{F}'))}\ [15]$$

$$\frac{\begin{array}{c}\mathrm{T} \lhd \mathsf{SEQUENCE}\, (\varphi' :: \Phi') \\ \varphi' \lhd \mathsf{Field}\, (l', \tau', \mathrm{T}', \sigma', s') \qquad \tilde{V} \lhd \tilde{v}' :: \tilde{V}' \qquad \langle \mathcal{A} \rangle \Gamma \vdash \mathcal{D}(\tilde{v}') \sqsubseteq (\tau', \mathrm{T}') \\ \langle \mathcal{A} \rangle \Gamma \models \tilde{v}' : \tau'\, \mathrm{T}' \qquad \langle \mathcal{A} \rangle \Gamma \models (\psi, \mathsf{Const}\, \tilde{V}') : \tau\ (\mathsf{SEQUENCE}\, \Phi')\end{array}}{\langle \mathcal{A} \rangle \Gamma \models (\psi, \mathsf{Const}\, \tilde{V}) : \tau\, \mathrm{T}}\ [16]$$

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\, (l', \tau', \mathrm{T}, \sigma', s') \\ s' \lhd \mathsf{Some\ OPTIONAL} \qquad \langle \mathcal{A} \rangle \Gamma \models \tilde{v} : \tau\ (\mathsf{SEQUENCE}\, \Phi')\end{array}}{\langle \mathcal{A} \rangle \Gamma \models \tilde{v} : \tau\ (\mathsf{SEQUENCE}\, ([\varphi'] \sqcup \Phi'))}\ [17]$$

$$\frac{\mathrm{T} \lhd \mathsf{SEQUENCE}\, [\,] \qquad \psi \lhd (\mathsf{UNIVERSAL}, \mathsf{Imm}\, 16)}{\langle \mathcal{A} \rangle \Gamma \models (\psi, \mathsf{Const}\, [\,]) : [\mathsf{Tag}\ \psi\ \mathsf{IMPLICIT}]\ \mathrm{T}}\ [18]$$

$$\frac{\mathrm{T} \lhd \mathsf{SET}\,([\varphi'] \sqcup \Phi') \qquad \varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \qquad \tilde{V} \lhd [\tilde{v}'] \sqcup \tilde{V}' \qquad \langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}(\tilde{v}') \sqsubseteq (\tau', \mathrm{T}') \qquad \langle\mathcal{A}\rangle\Gamma \models \tilde{v}' : \tau'\,\mathrm{T}' \qquad \langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Const}\,\tilde{V}') : \tau\ (\mathsf{SET}\,\Phi')}{\langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Const}\,\tilde{V}) : \tau\,\mathrm{T}}\ [19]$$

$$\frac{\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}, \sigma', s') \qquad s' \lhd \mathsf{Some\,OPTIONAL} \qquad \langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{SET}\,\Phi')}{\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{SET}\,([\varphi'] \sqcup \Phi'))}\ [20]$$

$$\frac{\mathrm{T} \lhd \mathsf{SET}\,[\,] \qquad \psi \lhd (\mathsf{UNIVERSAL}, \mathsf{Imm}\,17)}{\langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Const}\,[\,]) : [\mathsf{Tag}\ \psi\ \mathsf{IMPLICIT}]\ \mathrm{T}}\ [21]$$

8.3 Types canoniques

2

$$\frac{c \not\lhd \mathsf{UNIVERSAL} \qquad n \geqslant 0 \qquad \langle\mathcal{A}\rangle\Gamma, x \vdash_{\overline{c}} \tau'\,\mathrm{T}}{\langle\mathcal{A}\rangle\Gamma, x \vdash_{\overline{c}} ((\mathsf{Tag}\ (c, \mathsf{Imm}\,n)\ \mathsf{EXPLICIT}) :: \tau')\ \mathrm{T}}\ [1]$$

$$\frac{x' \in \mathcal{A} \qquad \Gamma(x') \lhd (\alpha', \tau', \{(\mathrm{T}', \sigma')\}) \qquad \langle\mathcal{A}\rangle\Gamma, x' \vdash_{\overline{c}} \tau'\,\mathrm{T}'}{\langle\mathcal{A}\rangle\Gamma, x \vdash_{\overline{c}} [\,]\ (\mathsf{TRef}\,x')}\ [2]$$

$$\frac{\tau \lhd [\mathsf{Tag}\ (\mathsf{UNIVERSAL}, \mathsf{Imm}\,1)\ \mathsf{IMPLICIT}]}{\langle\mathcal{A}\rangle\Gamma, x \vdash_{\overline{c}} \tau\ \mathsf{BOOLEAN}}\ [3]$$

$$\frac{\tau \lhd [\mathsf{Tag}\ (\mathsf{UNIVERSAL}, \mathsf{Imm}\,2)\ \mathsf{IMPLICIT}]}{\langle\mathcal{A}\rangle\Gamma, x \vdash_{\overline{c}} \tau\ (\mathsf{INTEGER}\ \ \_)}\ [4]$$

$$\frac{\tau \lhd [\mathsf{Tag}\ (\mathsf{UNIVERSAL}, \mathsf{Imm}\,3)\ \mathsf{IMPLICIT}]}{\langle\mathcal{A}\rangle\Gamma, x \vdash_{\overline{c}} \tau\ (\mathsf{BIT\ STRING}\ \ \_)}\ [5]$$

$$\frac{\tau \lhd [\mathsf{Tag}\ (\mathsf{UNIVERSAL}, \mathsf{Imm}\,4)\ \mathsf{IMPLICIT}]}{\langle\mathcal{A}\rangle\Gamma, x \vdash_{\overline{c}} \tau\ \mathsf{OCTET\ STRING}}\ [6]$$

$$\frac{\tau \lhd [\mathsf{Tag}\ (\mathsf{UNIVERSAL}, \mathsf{Imm}\,5)\ \mathsf{IMPLICIT}]}{\langle\mathcal{A}\rangle\Gamma, x \vdash_{\overline{c}} \tau\ \mathsf{NULL}}\ [7]$$

$$\frac{\tau \lhd [\mathsf{Tag}\ (\mathsf{UNIVERSAL}, \mathsf{Imm}\,9)\ \mathsf{IMPLICIT}]}{\langle\mathcal{A}\rangle\Gamma, \text{"REAL"} \vdash_{\overline{c}} \tau\ \mathrm{T}}\ [8]$$

$$\frac{\tau \lhd [\mathsf{Tag}\ (\mathsf{UNIVERSAL}, \mathsf{Imm}\, 10)\ \mathsf{IMPLICIT}]}{\langle \mathcal{A} \rangle \Gamma, x \vdash_{\mathcal{C}} \tau\ (\mathsf{ENUMERATED}\ \ \_)}\ [9]$$

$$\frac{\tau \lhd [\mathsf{Tag}\ (\mathsf{UNIVERSAL}, \mathsf{Imm}\, 18)\ \mathsf{IMPLICIT}]}{\langle \mathcal{A} \rangle \Gamma, x \vdash_{\mathcal{C}} \tau\ (\mathsf{CharString}\, \mathsf{Numeric})}\ [10]$$

$$\frac{\tau \lhd [\mathsf{Tag}\ (\mathsf{UNIVERSAL}, \mathsf{Imm}\, 19)\ \mathsf{IMPLICIT}]}{\langle \mathcal{A} \rangle \Gamma, x \vdash_{\mathcal{C}} \tau\ (\mathsf{CharString}\, \mathsf{Printable})}\ [11]$$

$$\frac{\tau \lhd [\mathsf{Tag}\ (\mathsf{UNIVERSAL}, \mathsf{Imm}\, 20)\ \mathsf{IMPLICIT}]}{\langle \mathcal{A} \rangle \Gamma, x \vdash_{\mathcal{C}} \tau\ (\mathsf{CharString}\, (\mathsf{Teletex} \mid \mathsf{T}_6 1))}\ [12]$$

$$\frac{\tau \lhd [\mathsf{Tag}\ (\mathsf{UNIVERSAL}, \mathsf{Imm}\, 21)\ \mathsf{IMPLICIT}]}{\langle \mathcal{A} \rangle \Gamma, x \vdash_{\mathcal{C}} \tau\ (\mathsf{CharString}\, \mathsf{Videotex})}\ [13]$$

$$\frac{\tau \lhd [\mathsf{Tag}\ (\mathsf{UNIVERSAL}, \mathsf{Imm}\, 22)\ \mathsf{IMPLICIT}]}{\langle \mathcal{A} \rangle \Gamma, x \vdash_{\mathcal{C}} \tau\ (\mathsf{CharString}\, \mathsf{IA}_5)}\ [14]$$

$$\frac{\tau \lhd [\mathsf{Tag}\ (\mathsf{UNIVERSAL}, \mathsf{Imm}\, 25)\ \mathsf{IMPLICIT}]}{\langle \mathcal{A} \rangle \Gamma, x \vdash_{\mathcal{C}} \tau\ (\mathsf{CharString}\, \mathsf{Graphic})}\ [15]$$

$$\frac{\tau \lhd [\mathsf{Tag}\ (\mathsf{UNIVERSAL}, \mathsf{Imm}\, 26)\ \mathsf{IMPLICIT}]}{\langle \mathcal{A} \rangle \Gamma, x \vdash_{\mathcal{C}} \tau\ (\mathsf{CharString}\, (\mathsf{Visible} \mid \mathsf{ISO}_{646}))}\ [16]$$

$$\frac{\tau \lhd [\mathsf{Tag}\ (\mathsf{UNIVERSAL}, \mathsf{Imm}\, 27)\ \mathsf{IMPLICIT}]}{\langle \mathcal{A} \rangle \Gamma, x \vdash_{\mathcal{C}} \tau\ (\mathsf{CharString}\, \mathsf{General})}\ [17]$$

$$\frac{\tau \lhd [\mathsf{Tag}\ (\mathsf{UNIVERSAL}, \mathsf{Imm}\, 16)\ \mathsf{IMPLICIT}] \qquad \langle \mathcal{A} \rangle \Gamma, \text{""} \vdash_{\mathcal{C}} \tau'\ \mathrm{T}'}{\langle \mathcal{A} \rangle \Gamma, x \vdash_{\mathcal{C}} \tau\ (\mathsf{SEQUENCE\ OF}\, \tau'\, \mathrm{T}'\, \sigma')}\ [18]$$

$$\frac{\tau \lhd [\mathsf{Tag}\ (\mathsf{UNIVERSAL}, \mathsf{Imm}\, 17)\ \mathsf{IMPLICIT}] \qquad \langle \mathcal{A} \rangle \Gamma, x \vdash_{\mathcal{C}} \tau'\ \mathrm{T}'}{\langle \mathcal{A} \rangle \Gamma, x \vdash_{\mathcal{C}} \tau\ (\mathsf{SET\ OF}\, \tau'\, \mathrm{T}'\, \sigma')}\ [19]$$

$$\frac{\begin{array}{c} f' \lhd (l', \tau', \mathrm{T}', \sigma', s') \\ \langle \mathcal{A} \rangle \Gamma, \text{""} \vdash_{\mathcal{C}} \tau'\ \mathrm{T}' \qquad \langle \mathcal{A} \rangle \Gamma, x \vdash_{\mathcal{C}} [\,]\ (\mathsf{CHOICE}\, \mathcal{F}') \end{array}}{\langle \mathcal{A} \rangle \Gamma, x \vdash_{\mathcal{C}} [\,]\ (\mathsf{CHOICE}\, ([f'] \sqcup \mathcal{F}'))}\ [20]$$

$$\langle \mathcal{A} \rangle \Gamma, x \vdash_{\mathcal{C}} [\,]\ (\mathsf{CHOICE}\, [\,])\ [21]$$

$$\frac{\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \quad \langle\mathcal{A}\rangle\Gamma, \text{""} \vdash_{\overline{c}} \tau'\ \mathrm{T}' \qquad \langle\mathcal{A}\rangle\Gamma, x \vdash_{\overline{c}} \tau\ (\mathsf{SET}\,\Phi')}{\langle\mathcal{A}\rangle\Gamma, x \vdash_{\overline{c}} \tau\,(\mathsf{SET}\,([\varphi'] \sqcup \Phi'))}\ [22]$$

$$\frac{\tau' \lhd [\mathsf{Tag}\ (\mathsf{UNIVERSAL}, \mathsf{Imm}\,17)\ \mathsf{IMPLICIT}]}{\langle\mathcal{A}\rangle\Gamma, x \vdash_{\overline{c}} \tau\ (\mathsf{SET}\,[\,])}\ [23]$$

$$\frac{\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \quad \langle\mathcal{A}\rangle\Gamma, \text{""} \vdash_{\overline{c}} \tau'\,\mathrm{T}' \qquad \langle\mathcal{A}\rangle\Gamma, x \vdash_{\overline{c}} \tau\ (\mathsf{SEQUENCE}\,\Phi')}{\langle\mathcal{A}\rangle\Gamma, x \vdash_{\overline{c}} \tau\,(\mathsf{SEQUENCE}\,([\varphi'] \sqcup \Phi'))}\ [24]$$

$$\frac{\tau \lhd [\mathsf{Tag}\ (\mathsf{UNIVERSAL}, \mathsf{Imm}\,16)\ \mathsf{IMPLICIT}]}{\langle\mathcal{A}\rangle\Gamma, x \vdash_{\overline{c}} \tau\ (\mathsf{SEQUENCE}\,[\,])}\ [25]$$

8.4 Correction du codage

Dans les preuves qui suivent, si une hypothèse est une conjonction de propositions, celles-ci seront étiquetées de la forme $\mathbf{H}.x$. Sinon l'hypothèse sera **Hyp** et ses prémisses $\mathbf{H}.x$. Il en sera de même avec les conclusions, à partir du patron **C**. Les prémisses des diverses propositions seront étiquetées à l'avenant : $\mathbf{H}.x.y$, $\mathbf{C}.x.y$, etc.

De même que les relations introduites dans cette thèse sont toutes définies par des algorithmes, ce qui implique que les règles sont ordonnées, les preuves seront elles aussi présentées par cas ordonnés.

Dans la partition en cas de la preuve, certaines caractéristiques des cas seront parfois introduites comme hypothèses, dite *hypothèses de filtre* et notées $\mathbf{HF}.x$, pour être plus facilement référencées. Elles sont nécessaires pour la définition complète du cas courant. Cette présentation est notamment commode quand le cas est défini par une clause **when** dans un filtre de règle, et que la proposition ainsi introduite n'est pas seulement syntaxique (la clause **when** effectue des calculs).

Parmi les prémisses, certaines sont des extensions de l' environnement de la règle (en OCaml l'équivalent serait un **let ... in ...**). Elles seront distinguées par l'emploi du signe $\triangleq$.

Théorème 8.4.1 (Correction du codage)**.**

Soit Γ *un environnement de types canonique de domaine* $\mathcal{A}$
et $x \in \mathcal{A} \wedge \Gamma(x) \lhd (\alpha, \tau, \{(T, \sigma)\})$
Si $\langle\mathcal{A}\rangle\Gamma, x \vdash v : T$
et $\langle\mathcal{A}\rangle\Gamma \vdash v : \tau\ T \Rightarrow \tilde{v}$
alors $\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ T$

Preuve du théorème 8.4.1

On applique d'abord le lemme 8.4.2 puis le lemme 8.4.3. □

Lemme 8.4.2.

$$\begin{array}{ll} \mathit{Si} & \vdash_{47} (\langle\mathcal{A}_0\rangle\Gamma_0, \langle\mathcal{B}_0\rangle\Delta_0) \to (\langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta) \\ \mathit{et} & x \in \mathcal{A} \wedge \Gamma(x) \lhd (\alpha, \tau, \{(T, \sigma)\}) \\ \mathit{alors} & \langle\mathcal{A}\rangle\Gamma, x \vdash_{\overline{c}} \tau\ T \end{array}$$

Schéma de preuve du lemme 8.4.2

Étiquetons les propositions du lemme :

— **H.0** : $\vdash_{47} (\langle\mathcal{A}_0\rangle\Gamma_0, \langle\mathcal{B}_0\rangle\Delta_0) \to (\langle\mathcal{A}\rangle\Gamma, \langle\mathcal{B}\rangle\Delta)$

— **H.1** : $x \in \mathcal{A} \wedge \Gamma(x) \lhd (\alpha, \tau, \{(\mathrm{T}, \sigma)\})$

— **C** : $\langle\mathcal{A}\rangle\Gamma, x \vdash_{\overline{c}} \tau\ \mathrm{T}$

H.0 implique que l'un des environnements de type servant à construire $\langle\mathcal{A}\rangle\Gamma$ est produit par l'algorithme $\vdash_{\overline{67}}$ (section 6.10.2 page 139). Nommons-le $\langle\mathcal{A}'\rangle\Gamma'$. Les calculs de $\vdash_{47}$ ne retranchant jamais de liaisons, on a : $x \in \mathcal{A}' \wedge \Gamma'(x) \lhd (\alpha, \tau, \{(\mathrm{T}, \sigma)\})$. On montre alors dans un premier temps que $\langle\mathcal{A}'\rangle\Gamma' \vdash_{\overline{c}} \tau\ \mathrm{T}$, puis on généralise à **C.1**. □

Lemme 8.4.3.

$$\begin{array}{ll} \mathit{Soit} & \langle\mathcal{A}\rangle\Gamma, x \vdash_{\overline{c}} \tau\ T \\ \mathit{Si} & \langle\mathcal{A}\rangle\Gamma, x \vdash v : T \\ \mathit{et} & \langle\mathcal{A}\rangle\Gamma \vdash v : \tau\ T \Rightarrow \tilde{v} \\ \mathit{alors} & \langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ T \end{array}$$

Preuve du lemme 8.4.3

Étiquetons les propositions de ce lemme :

— **H.1** : $\langle\mathcal{A}\rangle\Gamma, x \vdash_{\overline{c}} \tau\ \mathrm{T}$

— **H.2** : $\langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathrm{T}$

— **H.3** : $\langle\mathcal{A}\rangle\Gamma \vdash v : \tau\ \mathrm{T} \Rightarrow \tilde{v}$

— **C** : $\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ \mathrm{T}$

Posons :

— **Hyp** $\triangleq$ **H.1** $\wedge$ **H.2** $\wedge$ **H.3**

— **PA** : $\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}(\tilde{v}) \sqsubseteq (\tau, \mathrm{T})$

La preuve de **Hyp** $\Rightarrow$ **C** (notre lemme) nécessite dans certains cas le théorème **Hyp** $\Rightarrow$ **PA** (qui s'applique dans toute sa généralité, c'est-à-dire pour tout **H**). Nous pourrions donc prouver au préalable **Hyp** $\Rightarrow$ **PA**, et ensuite nous servir de ce résultat liminaire dans les cas de la preuve de notre lemme qui le nécessitent. Cette façon de procéder est techniquement la plus simple, mais elle implique la duplication des nombreuses

hypothèses constituant **Hyp** et de leurs nombreuses prémisses. Pour ne pas remplir trop de pages redondantes, nous allons procéder d'une façon indirecte qui mène au même résultat tout en permettant le partage des hypothèses de **Hyp**.

Nous allons donc, pour prouver notre lemme, démontrer un résultat plus général qui sera notre *unique* hypothèse d'induction :

— $\mathbf{Hyp} \Rightarrow (\mathbf{PA} \wedge \mathbf{C})$

Dans tous les cas, on commencera par prouver dans un premier temps

— $\mathbf{Hyp} \Rightarrow \mathbf{PA}$

Puis on prouvera

$$\begin{cases} \mathbf{Hyp} \Rightarrow \mathbf{C} \\ \textit{ou} \\ \mathbf{Hyp} \Rightarrow (\mathbf{PA} \Rightarrow \mathbf{C}) \end{cases}$$

Le premier sous-cas correspond aux cas où **PA** n'est pas nécessaire (en tout cas explicitement) pour prouver **C**. Donc dans tous les cas on démontre bien

— $\mathbf{Hyp} \Rightarrow (\mathbf{PA} \wedge \mathbf{C})$

Nous allons raffiner **PA** (*Proposition Auxiliaire*), pour y faire apparaître la définition par cas de $\mathcal{D}$. Par définition (section 8.2 page 161) :

$$\langle \mathcal{A} \rangle \Gamma \vdash \mathcal{D}(\tilde{v}) \sqsubseteq (\tau, \mathrm{T}) \textit{ est équivalent à } \vdash_{\mathcal{D}} \tilde{v} \to e \Rightarrow \langle \mathcal{A} \rangle \Gamma \vdash e \sqsubseteq (\tau, \mathrm{T}).$$

Étiquetons :

— **PA.H** : $\vdash_{\mathcal{D}} \tilde{v} \to e$

— **PA.C** : $\langle \mathcal{A} \rangle \Gamma \vdash e \sqsubseteq (\tau, \mathrm{T})$

Pour résumer, voici la forme de la preuve. Supposons

— **H.1** : $\langle \mathcal{A} \rangle \Gamma, x \vdash_{\mathcal{C}} \tau\ \mathrm{T}$

— **H.2** : $\langle \mathcal{A} \rangle \Gamma, x \vdash v : \mathrm{T}$

— **H.3** : $\langle \mathcal{A} \rangle \Gamma \vdash v : \tau\ \mathrm{T} \Rightarrow \tilde{v}$

Alors, successivement

1. En supposant **Hyp** et **PA.H**, démontrons **PA.C**.
2. $$\begin{cases} \text{Démontrons } \mathbf{C}. \\ \textit{ou} \\ \text{En supposant } \mathbf{PA.H} \text{ et } \mathbf{PA.C} \text{ démontrons } \mathbf{C}. \end{cases}$$

La conjonction de ces deux démonstrations successives implique trivialement

— $\mathbf{Hyp} \Rightarrow (\mathbf{PA} \wedge \mathbf{C})$

qui est l'hypothèse d'induction et notre théorème dont on déduit alors trivialement notre lemme 8.4.3 page précédente (par simple élimination de **PA** à droite de l'implication). On terminera les deux sous-preuves par le symbole $\Diamond$, chaque cas par $\triangle$, et la preuve entière par $\square$. Nous

ne présenterons en détail que les cas les plus difficiles de cette longue preuve.

— $\tau \lhd (\mathsf{Tag}\ ((c, \mathsf{Imm}\, n)\, \mathbf{as}\, \psi)\ \mathsf{EXPLICIT}) :: \tau'$. Étiquetons cette hypothèse de filtre :
  — **HF.0** : $\tau \lhd (\mathsf{Tag}\ ((c, \mathsf{Imm}\, n)\, \mathbf{as}\, \psi)\ \mathsf{EXPLICIT}) :: \tau'$

  Les autres hypothèses sont :
  — **H.1** : $\langle \mathcal{A} \rangle \Gamma, x \models_{\overline{c}} \tau\, \mathrm{T}$

  La règle [1] page 166 peut s'appliquer :

$$\frac{c \not\lhd \mathsf{UNIVERSAL} \qquad n \geqslant 0 \qquad \langle \mathcal{A} \rangle \Gamma, x \models_{\overline{c}} \tau'\, \mathrm{T}}{\langle \mathcal{A} \rangle \Gamma, x \models_{\overline{c}} ((\mathsf{Tag}\ (c, \mathsf{Imm}\, n)\ \mathsf{EXPLICIT}) :: \tau')\ \mathrm{T}}\ [1]$$

  Appliquons directement la règle (i.e. toutes les variables de la règle sont liées aux variables de même nom dans l'environnement du cas courant de la preuve). Étiquetons ses prémisses :
  — **H.1.0** : $c \not\lhd \mathsf{UNIVERSAL}$
  — **H.1.1** : $n \geqslant 0$
  — **H.1.2** : $\langle \mathcal{A} \rangle \Gamma, x' \models_{\overline{c}} \tau'\, \mathrm{T}$

— **H.2** : $\langle \mathcal{A} \rangle \Gamma, x \vdash v : \mathrm{T}$
— **H.3** : $\langle \mathcal{A} \rangle \Gamma \vdash v : \tau\, \mathrm{T} \Rightarrow \tilde{v}$. C'est la règle [1] page 159 qui s'applique :

$$\frac{\langle \mathcal{A} \rangle \Gamma \vdash v : \tau'\, \mathrm{T} \Rightarrow \tilde{v}'}{\langle \mathcal{A} \rangle \Gamma \vdash v : ((\mathsf{Tag}\ \psi\ \mathsf{EXPLICIT}) :: \tau')\, \mathrm{T} \Rightarrow (\psi, \mathsf{Const}\, [\tilde{v}'])}\ [1]$$

  Appliquons directement la règle. Étiquetons alors sa prémisse :
  — **H.3.0** : $\langle \mathcal{A} \rangle \Gamma \vdash v : \tau'\, \mathrm{T} \Rightarrow \tilde{v}'$

  et le jugement résultant de la projection du champ inféré :
  — **H.3.1** : $(\psi, \mathsf{Const}\, [\tilde{v}']) \lhd \tilde{v}$

Reprenons l'énoncé du théorème et effectuons-y les renommages suivants :
— $\tau \xleftarrow{\alpha} \tau'$
— $\tilde{v} \xleftarrow{\alpha} \tilde{v}'$

Nous obtenons alors :

$$\begin{array}{ll} \mathit{Soit} & \langle \mathcal{A} \rangle \Gamma, x \models_{\overline{c}} \tau'\, \mathrm{T} \\ \mathit{Si} & \langle \mathcal{A} \rangle \Gamma, x \vdash v : \mathrm{T} \\ \mathit{et} & \langle \mathcal{A} \rangle \Gamma \vdash v : \tau'\, \mathrm{T} \Rightarrow \tilde{v}' \\ \mathit{alors} & \langle \mathcal{A} \rangle \Gamma \vdash \mathcal{D}\, (\tilde{v}') \sqsubseteq (\tau', \mathrm{T}) \\ \mathit{et} & \langle \mathcal{A} \rangle \Gamma \models \tilde{v}' : \tau'\, \mathrm{T} \end{array}$$

Nous reconnaissons en cet énoncé une instance de l'hypothèse d'induction, car ses prémisses sont, dans l'ordre d'écriture, **H.1.2**, **H.2** et **H.3.0**. En l'appliquant il vient donc :

— **H.4** : $\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}') \sqsubseteq (\tau', \mathrm{T})$
— **H.5** : $\langle\mathcal{A}\rangle\Gamma \models \tilde{v}' : \tau'\,\mathrm{T}$
Maintenant

1. Prouvons **Hyp** $\Rightarrow$ **PA**. On se donne :
   — **PA.H** : $\vdash_{\mathcal{D}} \tilde{v} \to e$.
   H.1.0 implique
   — is_built_in_tag (ψ) = **false**
   Par conséquent, en accord avec **H.3.1**, la règle [3] page 164 s'applique :

$$\frac{\vdash_{\mathcal{D}} \overline{v}' \to (\tau', \mathrm{T})}{\vdash_{\mathcal{D}} (\psi, \mathsf{Const}\,[\overline{v}']) \to ((\mathsf{Tag}\ \psi\ \mathsf{EXPLICIT}) :: \tau', \mathrm{T})}\ [3]$$

   Appliquons directement cette règle, après le renommage
   — $\mathrm{T} \xleftarrow{\alpha} \mathrm{T}_0$
   et étiquetons ensuite sa prémisse :
   — **PA.H.0** : $\vdash_{\mathcal{D}} \tilde{v}' \to (\tau'_0, \mathrm{T}_0)$
   et le jugement résultant de la projection du champ inféré :
   — **PA.H.1** : $((\mathsf{Tag}\ \psi\ \mathsf{EXPLICIT}) :: \tau'_0, \mathrm{T}_0) \lhd e$
   Prouvons alors :
   — **PA.C** : $\langle\mathcal{A}\rangle\Gamma \vdash e \sqsubseteq (\tau, \mathrm{T})$. Les propositions **HF.0** et **PA.H.1** font que la règle [4] page 161 est applicable :

$$\frac{\langle\mathcal{A}\rangle\Gamma \vdash (\tau'_0, \mathrm{T}_0) \sqsubseteq (\tau'_1, \mathrm{T}_1)}{\langle\mathcal{A}\rangle\Gamma \vdash (t :: \tau'_0, \mathrm{T}_0) \sqsubseteq (t :: \tau'_1, \mathrm{T}_1)}\ [4]$$

   Appliquons-là. Ce faisant nous effectuons les substitutions suivantes sur la règle (les autres consistant à substituer les termes liés dans l'environnement courant du cas de la preuve aux termes *de même nom* dans la règle) :
   — $t \xleftarrow{\beta} \mathsf{Tag}\ \psi\ \mathsf{EXPLICIT}$
   — $\tau'_1 \xleftarrow{\beta} \tau'$
   — $\mathrm{T}_1 \xleftarrow{\beta} \mathrm{T}$
   Étiquetons ensuite sa prémisse :
   — **PA.C.1** : $\langle\mathcal{A}\rangle\Gamma \vdash (\tau'_0, \mathrm{T}_0) \sqsubseteq (\tau', \mathrm{T})$
   D'autre part **PA.H.0** est équivalent à :
   — **H.6** : $\mathcal{D}\,(\tilde{v}') \triangleq (\tau'_0, \mathrm{T}_0)$
   Substituons alors $\mathcal{D}\,(\tilde{v}')$ par sa définition **H.6** dans **H.4** :
   — **H.7** : $\langle\mathcal{A}\rangle\Gamma \vdash (\tau'_0, \mathrm{T}_0) \sqsubseteq (\tau', \mathrm{T})$
   H.7 implique **PA.C.1**, et **PA.C.1** implique par définition **PA.C**. $\Diamond$
2. Prouvons **Hyp** $\Rightarrow$ **C**. On a :

— $\mathbf{C} : \langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\,\mathrm{T}$

H.3.1 permet d'envisager l'application directe de la règle [1] page 164 :

$$\frac{\langle\mathcal{A}\rangle\Gamma \models \tilde{v}' : \tau'\,\mathrm{T}}{\langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Const}\,[\tilde{v}']) : ((\mathsf{Tag}\ \psi\ \mathsf{EXPLICIT}) :: \tau')\,\mathrm{T}}\ [1]$$

Étiquetons sa prémisse :

— $\mathbf{C.0} : \langle\mathcal{A}\rangle\Gamma \models \tilde{v}' : \tau'\,\mathrm{T}$

On a : $\mathbf{H.5} = \mathbf{C.0}$, et $\mathbf{C.0}$ implique, par définition, $\mathbf{C}$. ◊

Conclusion : $\left.\begin{array}{l}\mathbf{Hyp} \Rightarrow \mathbf{PA}\\ \mathbf{Hyp} \Rightarrow \mathbf{C}\end{array}\right\}$ impliquent $\mathbf{Hyp} \Rightarrow (\mathbf{PA} \wedge \mathbf{C})$. △

— $\mathrm{T} \lhd \mathsf{TRef}\,(x')$ et $\tau \lhd [\,]$. Les autres hypothèses sont

— $\mathbf{H.1} : \langle\mathcal{A}\rangle\Gamma, x \vdash_{\overline{c}} \tau\,\mathrm{T}$. C'est la règle [2] page 166 qui s'applique :

$$\frac{x' \in \mathcal{A} \qquad \Gamma(x') \lhd (\alpha', \tau', \{(\mathrm{T}', \sigma')\}) \qquad \langle\mathcal{A}\rangle\Gamma, x' \vdash_{\overline{c}} \tau'\,\mathrm{T}'}{\langle\mathcal{A}\rangle\Gamma, x \vdash_{\overline{c}} [\,]\ (\mathsf{TRef}\,x')}\ [2]$$

Appliquons directement la règle (i.e. toutes les variables de la règle sont liées aux variables de même nom dans l'environnement du cas courant de la preuve). Étiquetons ses prémisses (nous avons regroupé les deux premières) :

— $\mathbf{H.1.0} : x' \in \mathcal{A} \wedge \Gamma(x') \lhd (\alpha', \tau', \{(\mathrm{T}', \sigma')\})$

— $\mathbf{H.1.1} : \langle\mathcal{A}\rangle\Gamma, x \vdash_{\overline{c}} \tau'\,\mathrm{T}'$

— $\mathbf{H.2} : \langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathrm{T}$. C'est la règle [1] page 144 qui s'applique :

$$\frac{x' \in \mathcal{A} \qquad \Gamma(x') \lhd (\alpha', \tau', \{(\mathrm{T}', \sigma')\}) \qquad \Gamma, x' \vdash v : \mathrm{T}'}{\langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathsf{TRef}\,(x')}\ [1]$$

Appliquons-là directement puis étiquetons ses prémisses (nous avons regroupé les deux premières) :

— $\mathbf{H.2.0} : x' \in \mathcal{A} \wedge \Gamma(x') \lhd (\alpha', \tau', \{(\mathrm{T}', \sigma')\})$

— $\mathbf{H.2.1} : \langle\mathcal{A}\rangle\Gamma, x' \vdash v : \mathrm{T}'$

— $\mathbf{H.3} : \langle\mathcal{A}\rangle\Gamma \vdash v : \tau\,\mathrm{T} \Rightarrow \tilde{v}$. C'est la règle [2] page 159 qui s'applique :

$$\frac{\begin{array}{c}x' \in \mathcal{A}\\ \Gamma(x') \lhd (\alpha', \tau', \{(\mathrm{T}', \sigma')\}) \qquad \langle\mathcal{A}\rangle\Gamma \vdash v : \tau'\,\mathrm{T}' \Rightarrow \tilde{v}\end{array}}{\langle\mathcal{A}\rangle\Gamma \vdash v : [\,]\ (\mathsf{TRef}\,x') \Rightarrow \tilde{v}}\ [2]$$

Appliquons-là directement puis étiquetons ses prémisses (nous avons regroupé les deux premières) :

— **H.3.0** : $x' \in \mathcal{A} \wedge \Gamma(x') \lhd (\alpha', \tau', \{(\mathrm{T}', \sigma')\})$
— **H.3.1** : $\langle\mathcal{A}\rangle\Gamma \vdash v : \tau'\,\mathrm{T}' \Rightarrow \tilde{v}$

Reprenons l'énoncé du théorème et effectuons-y les renommages suivants :
— $\tau \xleftarrow{\alpha} \tau'$
— $\mathrm{T} \xleftarrow{\alpha} \mathrm{T}'$
Nous obtenons alors :

$$\begin{array}{ll} \mathit{Soit} & \langle\mathcal{A}\rangle\Gamma, x \vdash_{\mathrm{c}} \tau'\,\mathrm{T}' \\ \mathit{Si} & \langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathrm{T}' \\ \mathit{et} & \langle\mathcal{A}\rangle\Gamma \vdash v : \tau'\,\mathrm{T}' \Rightarrow \tilde{v} \\ \mathit{alors} & \langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}) \sqsubseteq (\tau', \mathrm{T}') \\ \mathit{et} & \langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau'\,\mathrm{T}' \end{array}$$

Nous reconnaissons en cet énoncé une instance de l'hypothèse d'induction, car ses prémisses sont, dans l'ordre d'écriture, **H.1.1**, **H.2.1** et **H.3.1**. En l'appliquant il vient donc :
— **H.4** : $\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}) \sqsubseteq (\tau', \mathrm{T}')$
— **H.5** : $\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau'\,\mathrm{T}'$
Maintenant

1. Prouvons **Hyp** $\Rightarrow$ **PA**.
   — **PA** : $\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}) \sqsubseteq (\tau, \mathrm{T})$. C'est la règle [3] page 161 qui s'applique :

$$\frac{x' \in \mathcal{A} \qquad \Gamma(x') \lhd (\alpha', \tau', \{(\mathrm{T}', \sigma')\}) \qquad \langle\mathcal{A}\rangle\Gamma \vdash e_0 \sqsubseteq (\tau', \mathrm{T}')}{\langle\mathcal{A}\rangle\Gamma \vdash e_0 \sqsubseteq ([\,], \mathsf{TRef}\,x')}\ [3]$$

   Appliquons-là. Ce faisant nous effectuons la substitution suivante sur la règle (les autres consistant à substituer les termes liés dans l'environnement courant du cas de la preuve aux termes *de même nom* dans la règle) :
   — $e_0 \xleftarrow{\beta} \mathcal{D}\,(\tilde{v})$
   Puis étiquetons les prémisses (nous avons regroupé les deux premières) :
   — **PA.0** : $x' \in \mathcal{A} \wedge \Gamma(x') \lhd (\alpha', \tau', \{(\mathrm{T}', \sigma')\})$
   — **PA.1** : $\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}) \sqsubseteq (\tau', \mathrm{T}')$
   Il est immédiat que **H.1.0** = **H.2.0** = **H.3.0** = **PA.0**. La proposition **H.4** implique **PA.1**. Les propositions **PA.0** et **PA.1** impliquent par définition **PA**. $\Diamond$

2. Prouvons **Hyp** $\Rightarrow$ **C**. On a :

— $\mathbf{C} : \langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\,\mathrm{T}$. C'est la règle [2] page 164 qui s'applique :

$$\frac{x' \in \mathcal{A} \qquad \Gamma(x') \lhd (\alpha', \tau', \{(\mathrm{T}', \sigma')\}) \qquad \langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau'\,\mathrm{T}'}{\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : [\,]\ (\mathsf{TRef}\,x')}\ [2]$$

Appliquons-là directement puis étiquetons ses prémisses (nous avons regroupé les deux premières) :
— $\mathbf{C.0} : x' \in \mathcal{A} \wedge \Gamma(x') \lhd (\alpha', \tau', \{(\mathrm{T}', \sigma')\})$
— $\mathbf{C.1} : \langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau'\,\mathrm{T}'$
Il est immédiat que $\mathbf{H.1.0} = \mathbf{H.2.0} = \mathbf{H.3.0} = \mathbf{C.0}$. D'autre part, $\mathbf{H.5} = \mathbf{C.1}$. Les propositions **C.0** et **C.1** impliquent par définition **C**. ◊

Conclusion : $\left.\begin{array}{l}\mathbf{Hyp} \Rightarrow \mathbf{PA} \\ \mathbf{Hyp} \Rightarrow \mathbf{C}\end{array}\right\}$ impliquent $\mathbf{Hyp} \Rightarrow (\mathbf{PA} \wedge \mathbf{C})$. △

— $\mathrm{T} \lhd \mathsf{SET\ OF}\,\tau'\,\mathrm{T}'\,\sigma'$
et $v \lhd \{[(\mathsf{None}, v')] \sqcup V'\}$
et $\tau \lhd [\mathsf{Tag}\ ((\mathsf{UNIVERSAL}, \mathsf{Imm}\,17)\,\mathbf{as}\,\psi)\ \mathsf{IMPLICIT}]$.
Distinguons des hypothèses de filtre :
— $\mathbf{HF.0} : \tau \lhd [\mathsf{Tag}\ ((\mathsf{UNIVERSAL}, \mathsf{Imm}\,17)\,\mathbf{as}\,\psi)\ \mathsf{IMPLICIT}]$
— $\mathbf{HF.1} : \mathrm{T} \lhd \mathsf{SET\ OF}\,\tau'\,\mathrm{T}'\,\sigma'$
Les autres hypothèses sont
— $\mathbf{H.1} : \langle\mathcal{A}\rangle\Gamma, x \vdash_{\overline{c}} \tau\,\mathrm{T}$. La règle [19] page 167 s'applique :

$$\frac{\tau \lhd [\mathsf{Tag}\ (\mathsf{UNIVERSAL}, \mathsf{Imm}\,17)\ \mathsf{IMPLICIT}] \qquad \langle\mathcal{A}\rangle\Gamma, x \vdash_{\overline{c}} \tau'\ \mathrm{T}'}{\langle\mathcal{A}\rangle\Gamma, x \vdash_{\overline{c}} \tau\ (\mathsf{SET\ OF}\,\tau'\,\mathrm{T}'\,\sigma')}\ [19]$$

Appliquons directement cette règle (i.e. toutes les variables de la règle sont liées aux variables de même nom dans l'environnement du cas courant de la preuve). Étiquetons sa prémisse non redondante (la première est en effet **HF.0**) :
— $\mathbf{H.1.0} : \langle\mathcal{A}\rangle\Gamma, x \vdash_{\overline{c}} \tau'\,\mathrm{T}'$
— $\mathbf{H.2} : \langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathrm{T}$. C'est la règle [11] page 145 qui s'applique :

$$\frac{\mathrm{T} \lhd (\mathsf{SEQUENCE\ OF} \mid \mathsf{SET\ OF})\ \tau'\,\mathrm{T}'\,\sigma' \qquad \langle\mathcal{A}\rangle\Gamma, \text{""} \vdash v' : \mathrm{T}' \qquad \langle\mathcal{A}\rangle\Gamma, x \vdash \{V'\} : \mathrm{T}}{\langle\mathcal{A}\rangle\Gamma, x \vdash \{[(\mathsf{None}, v')] \sqcup V'\} : \mathrm{T}}\ [11]$$

Appliquons directement cette règle et étiquetons ses prémisses non redondantes avec les hypothèses de filtre :

— **H.2.0** : $\langle\mathcal{A}\rangle\Gamma, "" \vdash v' : \mathrm{T}'$
— **H.2.1** : $\langle\mathcal{A}\rangle\Gamma, x \vdash \{V'\} : \mathrm{T}$
— **H.3** : $\langle\mathcal{A}\rangle\Gamma \vdash v : \tau\,\mathrm{T} \Rightarrow \tilde{v}$. C'est la règle [14] page 160 qui s'applique :

$$\frac{\begin{array}{c}\mathrm{T} \lhd \mathsf{SET\ OF}\,\tau'\,\mathrm{T}'\,\sigma' \qquad \langle\mathcal{A}\rangle\Gamma \vdash v' : \tau'\,\mathrm{T}' \Rightarrow \tilde{v}' \\ \langle\mathcal{A}\rangle\Gamma \vdash \{V'\} : \tau\,\mathrm{T} \Rightarrow (\psi, \mathsf{Const}\,\tilde{V}') \\ \tilde{v} \triangleq (\psi, \mathsf{Const}\,([\tilde{v}'] \sqcup \tilde{V}'))\end{array}}{\langle\mathcal{A}\rangle\Gamma \vdash \{[(\mathsf{None}, v')] \sqcup V'\} : \tau\,\mathrm{T} \Rightarrow \tilde{v}}\ [14]$$

Pour éviter une capture *a priori* du nom ψ (lié dans l'environnement courant du cas de la preuve par **HF.0**) lors de l'application, effectuons tout de suite le renommage suivant :
— $\psi \xleftarrow{\alpha} \overline{\psi}$
et appliquons alors la nouvelle règle directement. (Insistons sur le fait qu'il ne s'agissait pas d'une substitution de ψ, car la variable ψ n'est pas liée dans un champ projeté de la conclusion.) Étiquetons alors ses prémisses non redondantes avec les hypothèses de filtre :
— **H.3.0** : $\langle\mathcal{A}\rangle\Gamma \vdash v' : \tau'\,\mathrm{T}' \Rightarrow \tilde{v}'$
— **H.3.1** : $\langle\mathcal{A}\rangle\Gamma \vdash \{V'\} : \tau\,\mathrm{T} \Rightarrow (\overline{\psi}, \mathsf{Const}\,\tilde{V}')$
et aussi le jugement résultant de la projection du champ inféré :
— **H.3.2** : $(\overline{\psi}, \mathsf{Const}\,([\tilde{v}'] \sqcup \tilde{V}')) \lhd \tilde{v}$

Il est aisé de démontrer par induction le théorème suivant (considérer les règles [14] page 160 et [15] page 160 du codage) :

$$\begin{array}{ll} \textit{Soit} & \mathrm{T} \lhd \mathsf{SET\ OF}\,\tau'\,\mathrm{T}'\,\sigma' \\ \textit{et} & \tau \lhd [\mathsf{Tag}\ \psi\ \mathsf{IMPLICIT}] \\ \textit{Si} & \langle\mathcal{A}\rangle\Gamma \vdash v : \tau\,\mathrm{T} \Rightarrow (\overline{\psi}, \mathsf{Const}\,\tilde{V}) \\ \textit{alors} & \overline{\psi} = \psi \end{array}$$

Il découle de l'application de ce théorème :
— **H.4** : $\overline{\psi} = \psi$.
Maintenant

1. Prouvons **Hyp** $\Rightarrow$ **PA**. On se donne :
   — **PA.H** : $\vdash_{\overline{\mathcal{D}}} \tilde{v} \to e$. La proposition **HF.0** implique
   — is_built_in_type (ψ) = **true**
   Donc, en tenant compte aussi de la proposition **H.3.2** où ψ remplace $\overline{\psi}$ (comme le permet l'identité **H.4**), la règle [2]

page 164 s'applique :

$$\frac{\begin{array}{c}\text{is\_built\_in\_tag}\,(\psi)\\ \mathrm{T} \neq \mathsf{TRef}\ \_ \mid \_ < \_\_ \mid \mathsf{CHOICE}\ \_\end{array}}{\vdash_{\mathcal{D}} (\psi, \mathsf{Const}\_) \to ([\mathsf{Tag}\ \psi\ \mathsf{IMPLICIT}], \mathrm{T})}\ [2]$$

Appliquons directement cette règle, après le renommage
— $\mathrm{T} \xleftarrow{\alpha} \mathrm{T}_0$
Étiquetons alors sa prémisse non redondante :
— **PA.H.0** : $\mathrm{T}_0 \neq \mathsf{TRef}\ \_ \mid \_ < \_\_ \mid \mathsf{CHOICE}\ \_$
et le jugement résultant de la projection du chanmp inféré :
— **PA.H.1** : $([\mathsf{Tag}\ \psi\ \mathsf{IMPLICIT}], \mathrm{T}_0) \lhd e$
Prouvons alors :
— **PA.C** : $\langle\mathcal{A}\rangle\Gamma \vdash e \sqsubseteq (\tau, \mathrm{T})$. Les propositions **HF.0** et **HF.1** et **PA.H.0** et **PA.H.1** font que la règle [4] page 161 est applicable :

$$\frac{\langle\mathcal{A}\rangle\Gamma \vdash (\tau'_0, \mathrm{T}_0) \sqsubseteq (\tau'_1, \mathrm{T}_1)}{\langle\mathcal{A}\rangle\Gamma \vdash (t :: \tau'_0, \mathrm{T}_0) \sqsubseteq (t :: \tau'_1, \mathrm{T}_1)}\ [4]$$

Appliquons-là. Ce faisant nous effectuons les substitutions suivantes sur la règle (les autres consistant à substituer les termes liés dans l'environnement courant du cas de la preuve aux termes *de même nom* dans la règle) :
— $t \xleftarrow{\beta} \mathsf{Tag}\ \psi\ \mathsf{EXPLICIT}$
— $\tau'_0 \xleftarrow{\beta} [\,]$
— $\tau'_1 \xleftarrow{\beta} [\,]$
— $\mathrm{T}_1 \xleftarrow{\beta} \mathrm{T}$
Étiquetons ensuite sa prémisse :
— **PA.C.0** : $\langle\mathcal{A}\rangle\Gamma \vdash ([\,], \mathrm{T}_0) \sqsubseteq ([\,], \mathrm{T})$
Les propositions **PA.H.0** et **HF.1** impliquent **PA.C.0** par application de la règle [8] page 161 de la relation d'inclusion d'étiquetage

$$\frac{\begin{array}{l}\mathrm{T}_0 \not\lhd \mathsf{CHOICE}\ \_ \mid \mathsf{TRef}\ \_ \mid \_ < \_\_\\ \mathrm{T}_1 \not\lhd \mathsf{CHOICE}\ \_ \mid \mathsf{TRef}\ \_ \mid \_ < \_\_\end{array}}{\langle\mathcal{A}\rangle\Gamma \vdash ([\,], \mathrm{T}_0) \sqsubseteq ([\,], \mathrm{T}_1)}\ [8]$$

en effectuant la substitution $\mathrm{T}_1 \xleftarrow{\beta} \mathrm{T}$. Et **PA.C.0** implique par définition **PA.C**. ◊

2. Prouvons **Hyp** ⇒ **C**. On a :

— $\mathbf{C} : \langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\,\mathrm{T}$
La proposition **H.3.2** nous permet d'envisager l'application de la règle [12] page 165 :

$$\frac{\mathrm{T} \lhd (\mathsf{SET\ OF} \mid \mathsf{SEQUENCE\ OF})\,\tau'\,\mathrm{T}'\,\sigma' \qquad \langle\mathcal{A}\rangle\Gamma \models \tilde{v}' : \tau'\,\mathrm{T}' \qquad \langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Const}\,\tilde{V}') : \tau\,\mathrm{T}}{\langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Const}\,([\tilde{v}'] \sqcup \tilde{V}')) : \tau\,\mathrm{T}}\ [12]$$

Appliquons directement cette règle et étiquetons ses prémisses non redondantes avec les hypothèses de filtre :
— $\mathbf{C.0} : \langle\mathcal{A}\rangle\Gamma \models \tilde{v}' : \tau'\,\mathrm{T}'$
— $\mathbf{C.1} : \langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Const}\,\tilde{V}') : \tau\,\mathrm{T}$
Dans un premier temps reprenons l'énoncé du théorème et effectuons-y les renommages suivants :
— $\tau \xleftarrow{\alpha} \tau'$
— $\mathrm{T} \xleftarrow{\alpha} \mathrm{T}'$
— $\tilde{v} \xleftarrow{\alpha} \tilde{v}'$
Nous obtenons alors :

$$\begin{array}{ll} \mathit{Soit} & \langle\mathcal{A}\rangle\Gamma, x \vdash_{\overline{c}} \tau'\,\mathrm{T}' \\ \mathit{Si} & \langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathrm{T}' \\ \mathit{et} & \langle\mathcal{A}\rangle\Gamma \vdash v : \tau'\,\mathrm{T}' \Rightarrow \tilde{v}' \\ \mathit{alors} & \langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}') \sqsubseteq (\tau', \mathrm{T}') \\ \mathit{et} & \langle\mathcal{A}\rangle\Gamma \models \tilde{v}' : \tau'\,\mathrm{T}' \end{array}$$

Nous reconnaissons en cet énoncé une instance de l'hypothèse d'induction, car ses prémisses sont, dans l'ordre d'écriture, **H.1.0**, **H.2.0** et **H.3.0**. En l'appliquant il vient donc :
— $\mathbf{H.5} : \langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}') \sqsubseteq (\tau', \mathrm{T}')$
— $\mathbf{H.6} : \langle\mathcal{A}\rangle\Gamma \models \tilde{v}' : \tau'\,\mathrm{T}'$
On a immédiatement : **H.6** = **C.0**. Dans un deuxième temps, reprenons l'énoncé du théorème et effectuons-y les renommages suivants :
— $v \xleftarrow{\alpha} \{V'\}$
— $\tilde{v} \xleftarrow{\alpha} (\psi, \mathsf{Const}\,\tilde{V}')$
Nous obtenons alors :

$$\begin{array}{ll} \mathit{Soit} & \langle\mathcal{A}\rangle\Gamma, x \vdash_{\overline{c}} \tau\,\mathrm{T} \\ \mathit{Si} & \langle\mathcal{A}\rangle\Gamma, x \vdash \{V'\} : \mathrm{T} \\ \mathit{et} & \langle\mathcal{A}\rangle\Gamma \vdash \{V'\} : \tau\,\mathrm{T} \Rightarrow (\psi, \mathsf{Const}\,\tilde{V}') \\ \mathit{alors} & \langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\psi, \mathsf{Const}\,\tilde{V}') \sqsubseteq (\tau, \mathrm{T}) \\ \mathit{et} & \langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Const}\,\tilde{V}') : \tau\,\mathrm{T} \end{array}$$

Nous reconnaissons en cet énoncé une instance de l'hypothèse d'induction, car ses prémisses sont, dans l'ordre d'écriture,

H.1, **H.2.1** et **H.3.1** où ψ remplace $\overline{\psi}$ (comme le permet l'identité **H.4**). En l'appliquant il vient alors :
— **H.7** : $\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\psi, \mathsf{Const}\,\tilde{V}') \sqsubseteq (\tau, \mathrm{T})$
— **H.8** : $\langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Const}\,\tilde{V}') : \tau\,\mathrm{T}$
On a immédiatement : **H.8** = **C.1**. Finalement, **C.0** et **C.1** impliquent par définition **C**. ◊

Conclusion : $\left.\begin{array}{l}\mathbf{Hyp} \Rightarrow \mathbf{PA}\\ \mathbf{Hyp} \Rightarrow \mathbf{C}\end{array}\right\}$ impliquent $\mathbf{Hyp} \Rightarrow (\mathbf{PA} \wedge \mathbf{C})$. △

On note que **H.5** et **H.7** sont inutiles.

— $\mathrm{T} \lhd \mathsf{CHOICE}\,\mathcal{F}$ et $\mathcal{F} \lhd [f'] \sqcup \mathcal{F}'$
et $f' \lhd (l', \tau', \mathrm{T}', \sigma', s')$
et $v \lhd l' : v'$
et $\tau \lhd [\,]$.
Distinguons des hypothèses de filtre :
— **HF.0** : $f' \lhd (l', \tau', \mathrm{T}', \sigma', s')$
— **HF.1** : $\mathrm{T} \lhd \mathsf{CHOICE}\,([f'] \sqcup \mathcal{F}')$
Les autres hypothèses sont
— **H.1** : $\langle\mathcal{A}\rangle\Gamma, x \vdash_{\mathcal{C}} \tau\,\mathrm{T}$. La règle [20] page 167 qui peut s'appliquer :

$$\frac{\begin{array}{c}f' \lhd (l', \tau', \mathrm{T}', \sigma', s')\\ \langle\mathcal{A}\rangle\Gamma, \text{""} \vdash_{\mathcal{C}} \tau'\ \mathrm{T}' \qquad \langle\mathcal{A}\rangle\Gamma, x \vdash_{\mathcal{C}} [\,]\ (\mathsf{CHOICE}\,\mathcal{F}')\end{array}}{\langle\mathcal{A}\rangle\Gamma, x \vdash_{\mathcal{C}} [\,]\ (\mathsf{CHOICE}\,([f'] \sqcup \mathcal{F}'))}\ [20]$$

Appliquons directement la règle (i.e. toutes les variables de la règle sont liées aux variables de même nom dans l'environnement du cas courant de la preuve). Étiquetons sa prémisse :
— **H.1.0** : $\langle\mathcal{A}\rangle\Gamma, \text{""} \vdash_{\mathcal{C}} \tau'\,\mathrm{T}'$
— **H.2** : $\langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathrm{T}$. La règle [13] page 146 peut s'appliquer :

$$\frac{f' \lhd (l', \tau', \mathrm{T}', \sigma', s') \qquad \langle\mathcal{A}\rangle\Gamma, \text{""} \vdash v' : \mathrm{T}'}{\langle\mathcal{A}\rangle\Gamma, x \vdash (l' : v') : \mathsf{CHOICE}\,([f'] \sqcup \mathcal{F}')}\ [13]$$

Appliquons directement cette règle et étiquetons ses prémisses non redondantes avec les hypothèses de filtre :
— **H.2.0** : $\langle\mathcal{A}\rangle\Gamma, \text{""} \vdash v' : \mathrm{T}'$
— **H.3** : $\langle\mathcal{A}\rangle\Gamma \vdash v : \tau\,\mathrm{T} \Rightarrow \tilde{v}$. La règle [16] page 160 peut s'appliquer :

$$\frac{f' \lhd (l', \tau', \mathrm{T}', \sigma', s') \qquad \langle\mathcal{A}\rangle\Gamma \vdash v' : \tau'\,\mathrm{T}' \Rightarrow \tilde{v}}{\langle\mathcal{A}\rangle\Gamma \vdash (l' : v') : \tau\,(\mathsf{CHOICE}\,([f'] \sqcup \mathcal{F}')) \Rightarrow \tilde{v}}\ [16]$$

Appliquons directement cette règle et étiquetons une des prémisses :

— **H.3.0** : $\langle\mathcal{A}\rangle\Gamma \vdash v' : \tau'\,\mathrm{T}' \Rightarrow \tilde{v}$

Reprenons l'énoncé du théorème et effectuons-y les renommages suivants :
— $\tau \xleftarrow{\alpha} \tau'$
— $\mathrm{T} \xleftarrow{\alpha} \mathrm{T}'$
Nous obtenons alors :

$$\begin{array}{ll} \mathit{Soit} & \langle\mathcal{A}\rangle\Gamma, x \vdash_{\overline{c}} \tau'\,\mathrm{T}' \\ \mathit{Si} & \langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathrm{T}' \\ \mathit{et} & \langle\mathcal{A}\rangle\Gamma \vdash v : \tau'\,\mathrm{T}' \Rightarrow \tilde{v} \\ \mathit{alors} & \langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}) \sqsubseteq (\tau', \mathrm{T}') \\ \mathit{et} & \langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau'\,\mathrm{T}' \end{array}$$

Nous reconnaissons en cet énoncé une instance de l'hypothèse d'induction, car ses prémisses sont, dans l'ordre d'écriture, **H.1.0**, **H.2.0** et **H.3.0**. En l'appliquant il vient :
— **H.4** : $\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}) \sqsubseteq (\tau', \mathrm{T}')$
— **H.5** : $\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau'\,\mathrm{T}'$
Maintenant

1. Prouvons **Hyp** $\Rightarrow$ **PA**.
   — **PA** : $\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}) \sqsubseteq (\tau, \mathrm{T})$

   Les propositions **HF.1**, **HF.0** et **H.4** sont des instances possibles des prémisses de la règle [7] page 161 de la relation d'inclusion d'étiquetages

   $$\frac{\begin{array}{c}\mathrm{T} \lhd \mathsf{CHOICE}\,([f'] \sqcup \mathcal{F}') \\ f' \lhd (l', \tau', \mathrm{T}', \sigma', s') \qquad \langle\mathcal{A}\rangle\Gamma \vdash e_0 \sqsubseteq (\tau', \mathrm{T}')\end{array}}{\langle\mathcal{A}\rangle\Gamma \vdash e_0 \sqsubseteq ([\,], \mathrm{T})}\ [7]$$

   où on effectue le renommage $e_0 \xleftarrow{\alpha} \mathcal{D}\,(\tilde{v})$. Dérivons cette règle [7], après ce renommage, et la conclusion est précisément **PA**. ◊
2. Prouvons **Hyp** $\Rightarrow$ (**PA** $\Rightarrow$ **C**). On a :
   — **C** : $\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\,\mathrm{T}$
   La proposition **PA** nous permet d'envisager l'application de la règle [14] page 165 :

   $$\frac{\begin{array}{c}f' \lhd (l', \tau', \mathrm{T}', \sigma', s') \\ \langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}(\tilde{v}) \sqsubseteq (\tau', \mathrm{T}') \qquad \langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau'\,\mathrm{T}'\end{array}}{\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : [\,]\ (\mathsf{CHOICE}\,([f'] \sqcup \mathcal{F}'))}\ [14]$$

   Appliquons directement cette règle et étiquetons ses prémisses :

— **C.0** : $f' \lhd (l', \tau', \mathrm{T}', \sigma', s')$
— **C.1** : $\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}(\tilde{v}) \sqsubseteq (\tau', \mathrm{T}')$
— **C.2** : $\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau'\,\mathrm{T}'$

On a **HF.0** = **C.0**. On a évidemment : **PA** = **C.1**, et **H.5** = **C.2**. Enfin, **C.0** et **C.1** et **C.2** impliquent par définition **C**. ◊

Conclusion :
$\left.\begin{array}{l}\textbf{Hyp} \Rightarrow \textbf{PA}\\ \textbf{Hyp} \Rightarrow (\textbf{PA} \Rightarrow \textbf{C})\end{array}\right\}$ impliquent $\textbf{Hyp} \Rightarrow (\textbf{PA} \wedge \textbf{C})$. △

— $\mathrm{T} \lhd \mathsf{CHOICE}\,\mathcal{F}$
et $\mathcal{F} \lhd [f'] \sqcup \mathcal{F}'$
et $f' \lhd (l', \tau', \mathrm{T}', \sigma', s')$
et $\tau \lhd [\,]$.
Distinguons des hypothèses de filtre :
— **HF.0** : $\mathrm{T} \lhd \mathsf{CHOICE}\,([f'] \sqcup \mathcal{F}')$
— **HF.1** : $\tau \lhd [\,]$
— **HF.2** : $f' \lhd (l', \tau', \mathrm{T}', \sigma', s')$
Les autres hypothèses sont
— **H.1** : $\langle\mathcal{A}\rangle\Gamma, x \vdash_{\overline{c}} \tau\,\mathrm{T}$. C'est la règle [20] page 167 qui s'applique :

$$\frac{f' \lhd (l', \tau', \mathrm{T}', \sigma', s') \qquad \langle\mathcal{A}\rangle\Gamma, \text{""} \vdash_{\overline{c}} \tau'\ \mathrm{T}' \qquad \langle\mathcal{A}\rangle\Gamma, x \vdash_{\overline{c}} [\,]\ (\mathsf{CHOICE}\,\mathcal{F}')}{\langle\mathcal{A}\rangle\Gamma, x \vdash_{\overline{c}} [\,]\ (\mathsf{CHOICE}\,([f'] \sqcup \mathcal{F}'))}\ [20]$$

Appliquons directement cette règle (i.e. toutes les variables de la règle sont liées aux variables de même nom dans l'environnement du cas courant de la preuve). Étiquetons alors une des prémisses :
— **H.1.0** : $\langle\mathcal{A}\rangle\Gamma, x \vdash_{\overline{c}} [\,]\ (\mathsf{CHOICE}\,\mathcal{F}')$
— **H.2** : $\langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathrm{T}$. La règle [14] page 146 peut s'appliquer :

$$\frac{\langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathsf{CHOICE}\,\mathcal{F}'}{\langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathsf{CHOICE}\,([f'] \sqcup \mathcal{F}')}\ [14]$$

Appliquons directement cette règle et étiquetons sa prémisse :
— **H.2.0** : $\langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathsf{CHOICE}\,\mathcal{F}'$
— **H.3** : $\langle\mathcal{A}\rangle\Gamma \vdash v : \tau\,\mathrm{T} \Rightarrow \tilde{v}$. La règle [17] page 160 peut s'appliquer :

$$\frac{\langle\mathcal{A}\rangle\Gamma \vdash v : \tau\,(\mathsf{CHOICE}\,\mathcal{F}') \Rightarrow \tilde{v}}{\langle\mathcal{A}\rangle\Gamma \vdash v : \tau\,(\mathsf{CHOICE}\,([f'] \sqcup \mathcal{F}')) \Rightarrow \tilde{v}}\ [17]$$

Appliquons directement cette règle et étiquetons sa prémisse :
— **H.3.0** : $\langle\mathcal{A}\rangle\Gamma \vdash v : \tau\,(\mathsf{CHOICE}\,\mathcal{F}') \Rightarrow \tilde{v}$

Reprenons l'énoncé du théorème et effectuons-y le renommage suivant :
— $\mathrm{T} \xleftarrow{\alpha} \mathsf{CHOICE}\,\mathcal{F}'$
Nous obtenons alors :

$$\begin{array}{ll} \mathit{Soit} & \langle\mathcal{A}\rangle\Gamma, x \vdash_{\mathrm{C}} \tau\,(\mathsf{CHOICE}\,\mathcal{F}') \\ \mathit{Si} & \langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathsf{CHOICE}\,\mathcal{F}' \\ \mathit{et} & \langle\mathcal{A}\rangle\Gamma \vdash v : \tau\,(\mathsf{CHOICE}\,\mathcal{F}') \Rightarrow \tilde{v} \\ \mathit{alors} & \langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}) \sqsubseteq (\tau, \mathsf{CHOICE}\,\mathcal{F}') \\ \mathit{et} & \langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\,(\mathsf{CHOICE}\,\mathcal{F}') \end{array}$$

Nous reconnaissons en cet énoncé une instance de l'hypothèse d'induction, car ses prémisses sont, dans l'ordre d'écriture, **H.1.0** où $[\,]$ remplace τ (comme le permet **HF.1**), **H.2.0** et **H.3.0**. En l'appliquant il vient :
— **H.4** : $\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}) \sqsubseteq (\tau, \mathsf{CHOICE}\,\mathcal{F}')$
— **H.5** : $\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{CHOICE}\,\mathcal{F}')$
Maintenant

1. Prouvons **Hyp** $\Rightarrow$ **PA**.
   — **PA** : $\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}) \sqsubseteq (\tau, \mathrm{T})$. Pour déduire **H.4** nous pouvons appliquer la règle [7] page 161 :

$$\frac{\mathrm{T} \lhd \mathsf{CHOICE}\,([f'] \sqcup \mathcal{F}') \qquad f' \lhd (l', \tau', \mathrm{T}', \sigma', s') \qquad \langle\mathcal{A}\rangle\Gamma \vdash e_0 \sqsubseteq (\tau', \mathrm{T}')}{\langle\mathcal{A}\rangle\Gamma \vdash e_0 \sqsubseteq ([\,], \mathrm{T})}\ [7]$$

   Appliquons-là. Ce faisant nous effectuons les substitutions suivantes sur la règle (les autres consistant à substituer les termes liés dans l'environnement courant du cas de la preuve aux termes *de même nom* dans la règle) :
   — $e_0 \xleftarrow{\beta} \mathcal{D}\,(\tilde{v})$
   — $\mathcal{F} \xleftarrow{\beta} \mathcal{F}'$
   Puis étiquetons ses prémisses non redondantes avec les hypothèses de filtre :
   — **H.4.0** : $\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}) \sqsubseteq (\tau', \mathrm{T}')$
   Les propositions **HF.0**, **HF.2** et **H.4.0** sont des instances possibles des deux prémisses de la même règle [7] ci-dessus, après le renommage $e_0 \xleftarrow{\alpha} \mathcal{D}\,(\tilde{v})$. Cette dérivation (après ce renommage) engendre alors **PA**. $\Diamond$

2. Prouvons **Hyp** $\Rightarrow$ **C**. On a :
   — **C** : $\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\,\mathrm{T}$

La proposition **H.5** est la prémisse de la règle [15] page 165 :

$$\frac{\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{CHOICE}\,\mathcal{F}')}{\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{CHOICE}\,([f'] \sqcup \mathcal{F}'))}\ [15]$$

dont la conclusion est précisément **C**. ◊

Conclusion : $\left.\begin{array}{l}\mathbf{Hyp} \Rightarrow \mathbf{PA}\\ \mathbf{Hyp} \Rightarrow \mathbf{C}\end{array}\right\}$ impliquent $\mathbf{Hyp} \Rightarrow (\mathbf{PA} \wedge \mathbf{C})$. △

— $\mathrm{T} \lhd \mathsf{SEQUENCE}\,(\varphi' :: \Phi')$
et $\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s')$
et $v \lhd \{V\}$
et $V \lhd [(\mathsf{Some}\,l', v')] \sqcup V'$
et $\tau \lhd [\mathsf{Tag}\ \psi\ \mathsf{IMPLICIT}]$.
Distinguons deux hypothèses de filtre :
— **HF.0** : $\tau \lhd [\mathsf{Tag}\ \psi\ \mathsf{IMPLICIT}]$
— **HF.1** : $\mathrm{T} \lhd \mathsf{SEQUENCE}\,(\varphi' :: \Phi')$
Les autres hypothèses sont
— **H.1** : $\langle\mathcal{A}\rangle\Gamma, x \vdash_{\mathcal{C}} \tau\,\mathrm{T}$. Remarquons que $\Phi \lhd \varphi' :: \Phi'$ implique $\Phi \lhd [\varphi'] \sqcup \Phi'$, où φ' et Φ' sont identiques dans les deux jugements. Par conséquent, en tenant compte des hypothèses ci-dessus, nous pouvons envisager l'application de la règle [24] page 168 :

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s')\\ \langle\mathcal{A}\rangle\Gamma, \text{""} \vdash_{\mathcal{C}} \tau'\,\mathrm{T}' \qquad \langle\mathcal{A}\rangle\Gamma, x \vdash_{\mathcal{C}} \tau\ (\mathsf{SEQUENCE}\,\Phi')\end{array}}{\langle\mathcal{A}\rangle\Gamma, x \vdash_{\mathcal{C}} \tau\,(\mathsf{SEQUENCE}\,([\varphi'] \sqcup \Phi'))}\ [24]$$

Appliquons directement cette règle (i.e. toutes les variables de la règle sont liées aux variables de même nom dans l'environnement du cas courant de la preuve). Étiquetons ses prémisses non redondantes avec les hypothèses de filtre :
— **H.1.0** : $\langle\mathcal{A}\rangle\Gamma, x \vdash_{\mathcal{C}} \tau'\,\mathrm{T}'$
— **H.1.1** : $\langle\mathcal{A}\rangle\Gamma, x \vdash_{\mathcal{C}} \tau\ (\mathsf{SEQUENCE}\,\Phi')$
— **H.2** : $\langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathrm{T}$. Remarquons que $\Phi \lhd \varphi' :: \Phi'$ implique $\Phi \lhd [\varphi'] \sqcup \Phi'$, où φ' et Φ' sont identiques dans les deux jugements. Par conséquent, en tenant compte des hypothèses ci-dessus, nous pouvons envisager l'application de la règle [15] page 147 :

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \qquad V \lhd [(\mathsf{Some}\,l', v')] \sqcup V'\\ \langle\mathcal{A}\rangle\Gamma, \text{""} \vdash v' : \mathrm{T}' \qquad \langle\mathcal{A}\rangle\Gamma, x \vdash \{V'\} : \mathsf{SEQUENCE}\,\Phi'\end{array}}{\langle\mathcal{A}\rangle\Gamma, x \vdash \{V\} : \mathsf{SEQUENCE}\,([\varphi'] \sqcup \Phi')}\ [15]$$

Appliquons directement cette règle et étiquetons ses prémisses non redondantes avec les hypothèses de filtre :

— **H.2.0** : $\langle\mathcal{A}\rangle\Gamma, "" \vdash v' : \mathrm{T}'$
— **H.2.1** : $\langle\mathcal{A}\rangle\Gamma, x \vdash \{V'\} : \mathsf{SEQUENCE}\,\Phi'$
— **H.3** : $\langle\mathcal{A}\rangle\Gamma \vdash v : \tau\,\mathrm{T} \Rightarrow \tilde{v}$. Remarquons que $\Phi \lhd \varphi' :: \Phi'$ implique $\Phi \lhd [\varphi'] \sqcup \Phi'$, où φ' et Φ' sont identiques dans les deux jugements. Par conséquent, en tenant compte des hypothèses ci-dessus, nous pouvons envisager l'application de la règle [18] à la page 160 :

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \\ V \lhd [(\mathsf{Some}\,l', v')] \sqcup V' \qquad \langle\mathcal{A}\rangle\Gamma \vdash v' : \tau'\,\mathrm{T}' \Rightarrow \tilde{v}' \\ \langle\mathcal{A}\rangle\Gamma \vdash \{V'\} : \tau\,(\mathsf{SEQUENCE}\,\Phi') \Rightarrow (\psi, \mathsf{Const}\,\tilde{V}') \\ \tilde{v} \triangleq (\psi, \mathsf{Const}\,(\tilde{v}' :: \tilde{V}'))\end{array}}{\langle\mathcal{A}\rangle\Gamma \vdash \{V\} : \tau\,(\mathsf{SEQUENCE}\,(\varphi' :: \Phi')) \Rightarrow \tilde{v}}\ [18]$$

Pour éviter une capture *a priori* du nom ψ (lié dans l'environnement courant du cas de la preuve par **HF.0**) lors de l'application, effectuons tout de suite le renommage suivant :
— $\psi \xleftarrow{\alpha} \overline{\psi}$
et appliquons alors la nouvelle règle directement. (Insistons sur le fait qu'il ne s'agissait pas d'une substitution de ψ, car la variable ψ n'est pas liée dans un champ projeté de la conclusion.) Étiquetons alors ses prémisses non redondantes avec les hypothèses de filtre :
— **H.3.0** : $\langle\mathcal{A}\rangle\Gamma \vdash v' : \tau'\,\mathrm{T}' \Rightarrow \tilde{v}'$
— **H.3.1** : $\langle\mathcal{A}\rangle\Gamma \vdash \{V'\} : \tau\,(\mathsf{SEQUENCE}\,\Phi') \Rightarrow (\overline{\psi}, \mathsf{Const}\,\tilde{V}')$
et aussi le jugement résultant de la projection du champ inféré :
— **H.3.2** : $(\overline{\psi}, \mathsf{Const}\,(\tilde{v}' :: \tilde{V}')) \lhd \tilde{v}$

Il est aisé de démontrer par induction le théorème suivant (considérer les règle [18] page 160, [19] page 161 et [20] page 161 du codage) :

$$\begin{array}{ll} \mathit{Soit} & \mathrm{T} \lhd \mathsf{SEQUENCE}\,\Phi \\ \mathit{et} & \tau \lhd [\mathsf{Tag}\ \psi\ \mathsf{IMPLICIT}] \\ \mathit{Si} & \langle\mathcal{A}\rangle\Gamma \vdash v : \tau\,\mathrm{T} \Rightarrow (\overline{\psi}, \mathsf{Const}\,\tilde{V}) \\ \mathit{alors} & \overline{\psi} = \psi \end{array}$$

Il découle de l'application de ce théorème :
— **H.4** : $\overline{\psi} = \psi$
Maintenant

1. Prouvons **Hyp** $\Rightarrow$ **PA**. On se donne :
   — **PA.H** : $\vDash_{\mathcal{D}} \tilde{v} \to e$. La proposition **HF.0** implique

— is_built_in_type (ψ) = **true**
Donc, en tenant compte de **H.3.2** où ψ remplace $\overline{\psi}$ (comme le permet l'identité **H.4**), la règle [2] page 164 s'applique :

$$\frac{\begin{array}{c}\text{is\_built\_in\_tag}\,(\psi)\\ \mathrm{T} \neq \mathsf{TRef}\ \_ \mid \_ < \_\_ \mid \mathsf{CHOICE}\ \_\end{array}}{\vdash_{\mathcal{D}} (\psi, \mathsf{Const}\_) \to ([\mathsf{Tag}\ \psi\ \mathsf{IMPLICIT}], \mathrm{T})}\ [2]$$

Appliquons directement cette règle, après le renommage
— $\mathrm{T} \xleftarrow{\alpha} \mathrm{T}_0$
Étiquetons alors sa prémisse non redondante :
— **PA.H.0** : $\mathrm{T}_0 \neq \mathsf{TRef}\ \_ \mid \_ < \_\_ \mid \mathsf{CHOICE}\ \_$
et aussi le jugement résultant de la projection du champ inféré :
— **PA.H.1** : $([\mathsf{Tag}\ \psi\ \mathsf{IMPLICIT}], \mathrm{T}_0) \lhd e$

Prouvons alors :
— **PA.C** : $\langle\mathcal{A}\rangle\Gamma \vdash e \sqsubseteq (\tau, \mathrm{T})$. Les propositions **HF.0** et **HF.1** et **PA.H.0** et **PA.H.1** impliquent que c'est la règle [4] page 161 qui s'applique :

$$\frac{\langle\mathcal{A}\rangle\Gamma \vdash (\tau'_0, \mathrm{T}_0) \sqsubseteq (\tau'_1, \mathrm{T}_1)}{\langle\mathcal{A}\rangle\Gamma \vdash (t :: \tau'_0, \mathrm{T}_0) \sqsubseteq (t :: \tau'_1, \mathrm{T}_1)}\ [4]$$

Appliquons-là. Ce faisant nous effectuons sur la règle les substitutions suivantes :
— $\mathrm{T}_1 \xleftarrow{\beta} \mathrm{T}$
— $t \xleftarrow{\beta} \mathsf{Tag}\ \psi\ \mathsf{IMPLICIT}$
— $\tau_0 \xleftarrow{\beta} [\,]$
— $\tau_1 \xleftarrow{\beta} [\,]$
Puis étiquetons alors sa prémisse :
— **PA.C.0** : $\langle\mathcal{A}\rangle\Gamma \vdash ([\,], \mathrm{T}_0) \sqsubseteq ([\,], \mathrm{T})$
Or, **PA.C.0** peut être dérivée par application directe de la règle [8] page 161 de la relation d'inclusion d'étiquetage :

$$\frac{\begin{array}{l}\mathrm{T}_0 \not\lhd \mathsf{CHOICE}\ \_ \mid \mathsf{TRef}\ \_ \mid \_ < \_\_\\ \mathrm{T}_1 \not\lhd \mathsf{CHOICE}\ \_ \mid \mathsf{TRef}\ \_ \mid \_ < \_\_\end{array}}{\langle\mathcal{A}\rangle\Gamma \vdash ([\,], \mathrm{T}_0) \sqsubseteq ([\,], \mathrm{T}_1)}\ [8]$$

(ce faisant on effectue la substitution $\mathrm{T}_1 \xleftarrow{\beta} \mathrm{T}$), car **PA.H.0** et **HF.1** sont des instances des prémisses. Et **PA.C.0** implique par définition **PA.C**. ◊

2. Prouvons **Hyp** ⇒ **C**. On a :

— $\mathbf{C} : \langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\,\mathrm{T}$

La proposition **H.3.2** nous permet d'envisager l'application de la règle [16] page 165 :

$$\frac{\begin{array}{c}\mathrm{T} \lhd \mathsf{SEQUENCE}\,(\varphi' :: \Phi') \\ \varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \qquad \tilde{V} \lhd \tilde{v}' :: \tilde{V}' \\ \langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}(\tilde{v}') \sqsubseteq (\tau', \mathrm{T}') \qquad \langle\mathcal{A}\rangle\Gamma \models \tilde{v}' : \tau'\,\mathrm{T}' \\ \langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Const}\,\tilde{V}') : \tau\ (\mathsf{SEQUENCE}\,\Phi')\end{array}}{\langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Const}\,\tilde{V}) : \tau\,\mathrm{T}}\ [16]$$

Appliquons-là directement et étiquetons les prémisses non redondantes avec les hypothèses de filtre :

— $\mathbf{C.0} : \langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}(\tilde{v}') \sqsubseteq (\tau', \mathrm{T}')$

— $\mathbf{C.1} : \langle\mathcal{A}\rangle\Gamma \models \tilde{v}' : \tau'\,\mathrm{T}'$

— $\mathbf{C.2} : \langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Const}\,\tilde{V}') : \tau\ (\mathsf{SEQUENCE}\,\Phi')$

Dans un premier temps reprenons l'énoncé du théorème et effectuons-y les renommages suivants :

— $v \xleftarrow{\alpha} v'$

— $\tau \xleftarrow{\alpha} \tau'$

— $\mathrm{T} \xleftarrow{\alpha} \mathrm{T}'$

— $\tilde{v} \xleftarrow{\alpha} \tilde{v}'$

Nous obtenons alors :

$$\begin{array}{ll} \mathit{Soit} & \langle\mathcal{A}\rangle\Gamma, x \vdash_{\mathfrak{C}} \tau'\,\mathrm{T}' \\ \mathit{Si} & \langle\mathcal{A}\rangle\Gamma, x \vdash v' : \mathrm{T}' \\ \mathit{et} & \langle\mathcal{A}\rangle\Gamma \vdash v' : \tau'\,\mathrm{T}' \Rightarrow \tilde{v}' \\ \mathit{alors} & \langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}') \sqsubseteq (\tau', \mathrm{T}') \\ \mathit{et} & \langle\mathcal{A}\rangle\Gamma \models \tilde{v}' : \tau'\,\mathrm{T}' \end{array}$$

Nous reconnaissons en cet énoncé une instance de l'hypothèse d'induction, car ses prémisses sont, dans l'ordre d'écriture, **H.1.0**, **H.2.0** et **H.3.0**. En l'appliquant il vient

— $\mathbf{H.5} : \langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}') \sqsubseteq (\tau', \mathrm{T}')$

— $\mathbf{H.6} : \langle\mathcal{A}\rangle\Gamma \models \tilde{v}' : \tau'\ \mathrm{T}'$

Dans un deuxième temps, reprenons l'énoncé du théorème et effectuons-y les renommages suivants :

— $\mathrm{T} \xleftarrow{\alpha} \mathsf{SEQUENCE}\,\Phi'$

— $\tilde{v} \xleftarrow{\alpha} (\psi, \mathsf{Const}\,\tilde{V}')$

— $v \xleftarrow{\alpha} \mathbf{\{}V'\mathbf{\}}$

Nous obtenons alors :

$$\begin{array}{ll} \mathit{Soit} & \langle\mathcal{A}\rangle\Gamma, x \vdash_{\mathfrak{C}} \tau\,(\mathsf{SEQUENCE}\,\Phi') \\ \mathit{Si} & \langle\mathcal{A}\rangle\Gamma, x \vdash \mathbf{\{}V'\mathbf{\}} : \mathsf{SEQUENCE}\,\Phi' \\ \mathit{et} & \langle\mathcal{A}\rangle\Gamma \vdash \mathbf{\{}V'\mathbf{\}} : \tau\,(\mathsf{SEQUENCE}\,\Phi') \Rightarrow (\psi, \mathsf{Const}\,\tilde{V}') \\ \mathit{alors} & \langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\psi, \mathsf{Const}\,\tilde{V}') \sqsubseteq (\tau, \mathsf{SEQUENCE}\,\Phi') \\ \mathit{et} & \langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Const}\,\tilde{V}') : \tau\,(\mathsf{SEQUENCE}\,\Phi') \end{array}$$

Nous reconnaissons en cet énoncé une instance de l'hypothèse d'induction, car ses prémisses sont, dans l'ordre d'écriture, **H.1.1**, **H.2.1** et **H.3.1** (où $\overline{\psi} \xleftarrow{\beta} \psi$, comme le permet **H.4**). En l'appliquant il vient :
— **H.7** : $\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\psi, \mathsf{Const}\,\tilde{V}') \sqsubseteq (\tau, \mathsf{SEQUENCE}\,\Phi')$
— **H.8** : $\langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Const}\,\tilde{V}') : \tau\ (\mathsf{SEQUENCE}\,\Phi')$
On a : **H.5** = **C.0**, **H.6** = **C.1** et **H.8** = **C.2**. Les propositions **C.0**, **C.1** et **C.2** impliquent par définition **C**. ◊

Conclusion : $\left.\begin{array}{l}\mathbf{Hyp} \Rightarrow \mathbf{PA}\\ \mathbf{Hyp} \Rightarrow \mathbf{C}\end{array}\right\}$ impliquent $\mathbf{Hyp} \Rightarrow (\mathbf{PA} \wedge \mathbf{C})$. △

(On note que **H.7** est inutile.)
— $\mathrm{T} \lhd \mathsf{SEQUENCE}\,([\varphi'] \sqcup \Phi')$
et $\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s')$
et $s' \lhd \mathsf{Some}\,\mathsf{OPTIONAL}$
et $\tau \lhd [\mathsf{Tag}\ \psi\ \mathsf{IMPLICIT}]$.
Distinguons des hypothèses de filtre :
— **HF.0** : $\tau \lhd [\mathsf{Tag}\ \psi\ \mathsf{IMPLICIT}]$
— **HF.1** : $\mathrm{T} \lhd \mathsf{SEQUENCE}\,([\varphi'] \sqcup \Phi')$
Les autres hypothèses sont
— **H.1** : $\langle\mathcal{A}\rangle\Gamma, x \models_{\mathcal{C}} \tau\,\mathrm{T}$. La règle [24] page 168 peut s'appliquer :

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s')\\ \langle\mathcal{A}\rangle\Gamma, \text{""} \models_{\mathcal{C}} \tau'\,\mathrm{T}' \qquad \langle\mathcal{A}\rangle\Gamma, x \models_{\mathcal{C}} \tau\ (\mathsf{SEQUENCE}\,\Phi')\end{array}}{\langle\mathcal{A}\rangle\Gamma, x \models_{\mathcal{C}} \tau\,(\mathsf{SEQUENCE}\,([\varphi'] \sqcup \Phi'))}\ [24]$$

Appliquons directement cette règle (i.e. toutes les variables de la règle sont liées aux variables de même nom dans l'environnement du cas courant de la preuve). Étiquetons ses prémisses non redondantes avec les hypothèses de filtre :
— **H.1.0** : $\langle\mathcal{A}\rangle\Gamma, x \models_{\mathcal{C}} \tau'\,\mathrm{T}'$
— **H.1.1** : $\langle\mathcal{A}\rangle\Gamma, x \models_{\mathcal{C}} \tau\ (\mathsf{SEQUENCE}\,\Phi')$
— **H.2** : $\langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathrm{T}$. La règle [16] page 147 peut s'appliquer :

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s')\\ s' \lhd \mathsf{Some}\,\mathsf{OPTIONAL} \qquad \langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathsf{SEQUENCE}\,\Phi'\end{array}}{\langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathsf{SEQUENCE}\,([\varphi'] \sqcup \Phi')}\ [16]$$

Appliquons cette règle directement et étiquetons sa prémisse non redondante :
— **H.2.0** : $\langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathsf{SEQUENCE}\,\Phi'$

— **H.3** : $\langle\mathcal{A}\rangle\Gamma \vdash v : \tau\,\mathrm{T} \Rightarrow \tilde{v}$. La règle [19] page 161 peut s'appliquer

$$\frac{\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \qquad s' \lhd \mathsf{Some}\,\mathsf{OPTIONAL} \qquad \langle\mathcal{A}\rangle\Gamma \vdash v : \tau\,(\mathsf{SEQUENCE}\,\Phi') \Rightarrow \tilde{v}}{\langle\mathcal{A}\rangle\Gamma \vdash v : \tau\,(\mathsf{SEQUENCE}\,([\varphi'] \sqcup \Phi')) \Rightarrow \tilde{v}}\ [19]$$

Appliquons cette règle directement et étiquetons sa prémisse non redondante :

— **H.3.0** : $\langle\mathcal{A}\rangle\Gamma \vdash v : \tau\,(\mathsf{SEQUENCE}\,\Phi') \Rightarrow \tilde{v}$

Maintenant

1. Prouvons **Hyp** $\Rightarrow$ **PA**. On se donne :

   — **PA.H** : $\vdash_{\mathcal{D}} \tilde{v} \to e$. Il est aisé de prouver par induction le théorème suivant (considérer les règles [18] page 160, [19] page 161 et [20] page 161 du codage) :

   $$\begin{array}{ll} \textit{Soit} & \mathrm{T} \lhd \mathsf{SEQUENCE}\,\Phi \\ \textit{et} & \tau \lhd [\mathsf{Tag}\ \psi\ \mathsf{IMPLICIT}] \\ \textit{Si} & \langle\mathcal{A}\rangle\Gamma \vdash v : \tau\,\mathrm{T} \Rightarrow \tilde{v} \\ \textit{alors} & \tilde{v} \lhd (\overline{\psi}, \mathsf{Const}\ \_) \\ \textit{et} & \overline{\psi} = \psi \end{array}$$

   Il découle de l'application de ce théorème que la règle qui débute la preuve de **PA.H** est forcément [2] page 164 :

   $$\frac{\mathrm{is\_built\_in\_tag}\,(\psi) \qquad \mathrm{T} \neq \mathsf{TRef}\ \ \_ \mid \_ < \_\_ \mid \mathsf{CHOICE}\ \ \_}{\vdash_{\mathcal{D}} (\psi, \mathsf{Const}\_) \to ([\mathsf{Tag}\ \psi\ \mathsf{IMPLICIT}], \mathrm{T})}\ [2]$$

   Appliquons-là directement après le renommage :

   — $\mathrm{T} \xleftarrow{\alpha} \mathrm{T}_0$

   Étiquetons alors une prémisse utile pour la suite :

   — **PA.H.0** : $\mathrm{T}_0 \neq \mathsf{TRef}\ \ \_ \mid \_ < \_\_ \mid \mathsf{CHOICE}\ \ \_$

   et aussi le jugement résultant de la projection du champ inféré :

   — **PA.H.1** : $([\mathsf{Tag}\ \psi\ \mathsf{IMPLICIT}], \mathrm{T}_0) \lhd e$

   Prouvons alors :

   — **PA.C** : $\langle\mathcal{A}\rangle\Gamma \vdash e \sqsubseteq (\tau, \mathrm{T})$. Les propositions **HF.0**, **HF.1**, **PA.H.0** et **PA.H.1** impliquent que c'est la règle [4] page 161 qui s'applique :

   $$\frac{\langle\mathcal{A}\rangle\Gamma \vdash (\tau'_0, \mathrm{T}_0) \sqsubseteq (\tau'_1, \mathrm{T}_1)}{\langle\mathcal{A}\rangle\Gamma \vdash (t :: \tau'_0, \mathrm{T}_0) \sqsubseteq (t :: \tau'_1, \mathrm{T}_1)}\ [4]$$

   Appliquons-là. Ce faisant nous effectuons les substitutions suivantes :

— $t \xleftarrow{\beta} \mathsf{Tag}\ \psi\ \mathsf{IMPLICIT}$
— $\tau'_0 \xleftarrow{\beta} [\,]$
— $\tau'_1 \xleftarrow{\beta} [\,]$
— $\mathrm{T}_1 \xleftarrow{\beta} \mathrm{T}$
Sa prémisse est :
— **PA.C.0** : $\langle\mathcal{A}\rangle\Gamma \vdash ([\,], \mathrm{T}_0) \sqsubseteq ([\,], \mathrm{T})$
Les propositions **PA.H.0** et **HF.1** impliquent que **PA.C.0** est vraie, par application de la règle [8] page 161 de la relation d'inclusion d'étiquetage :

$$\frac{\begin{array}{l}\mathrm{T}_0 \not\lhd \mathsf{CHOICE}\ \ \_ \mid \mathsf{TRef}\ \ \_ \mid \_ < \_ \_ \\ \mathrm{T}_1 \not\lhd \mathsf{CHOICE}\ \ \_ \mid \mathsf{TRef}\ \ \_ \mid \_ < \_ \_\end{array}}{\langle\mathcal{A}\rangle\Gamma \vdash ([\,], \mathrm{T}_0) \sqsubseteq ([\,], \mathrm{T}_1)}\ [8]$$

où la substitutions suivante est ce faisant effectuée :
— $\mathrm{T}_1 \xleftarrow{\beta} \mathrm{T}$
Et **PA.C.0** implique par définition **PA.C**. ◊

2. Prouvons **Hyp** ⇒ **C**. On a :
— **C** : $\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ \mathrm{T}$
Les hypothèses de filtre permettent l'application de la règle [17] page 165 :

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}, \sigma', s') \\ s' \lhd \mathsf{Some}\,\mathsf{OPTIONAL} \qquad \langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{SEQUENCE}\,\Phi')\end{array}}{\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{SEQUENCE}\,([\varphi'] \sqcup \Phi'))}\ [17]$$

Appliquons-là directement et étiquetons sa prémisse non redondante avec les hypothèses de filtre :
— **C.0** : $\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{SEQUENCE}\,\Phi')$
Reprenons l'énoncé du théorème et effectuons-y le renommage suivant :
— $\mathrm{T} \xleftarrow{\alpha} \mathsf{SEQUENCE}\,\Phi'$
Nous obtenons alors :

$$\begin{array}{ll} \mathit{Soit} & \langle\mathcal{A}\rangle\Gamma, x \vdash_{\overline{c}} \tau\,(\mathsf{SEQUENCE}\,\Phi') \\ \mathit{Si} & \langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathsf{SEQUENCE}\,\Phi' \\ \mathit{et} & \langle\mathcal{A}\rangle\Gamma \vdash v : \tau\,(\mathsf{SEQUENCE}\,\Phi') \Rightarrow \tilde{v} \\ \mathit{alors} & \langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}) \sqsubseteq (\tau, \mathsf{SEQUENCE}\,\Phi') \\ \mathit{et} & \langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\,(\mathsf{SEQUENCE}\,\Phi') \end{array}$$

Nous reconnaissons en cet énoncé une instance de l'hypothèse d'induction, car ses prémisses sont, dans l'ordre d'écriture, **H.1.1**, **H.2.0** et **H.3.0**. En l'appliquant il vient :

— **H.4** : $\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}) \sqsubseteq (\tau, \mathsf{SEQUENCE}\,\Phi')$
— **H.5** : $\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{SEQUENCE}\,\Phi')$
On a : **H.5** = **C.0**, et **C.0** implique par définition **C**. $\Diamond$

Conclusion : $\left.\begin{array}{l}\mathbf{Hyp} \Rightarrow \mathbf{PA}\\ \mathbf{Hyp} \Rightarrow \mathbf{C}\end{array}\right\}$ impliquent $\mathbf{Hyp} \Rightarrow (\mathbf{PA} \wedge \mathbf{C})$. $\triangle$

On note que **H.4** est inutile.

— $\mathrm{T} \lhd \mathsf{SET}\,(\varphi' :: \Phi')$ et $\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s')$
et $v \lhd \{V\}$
et $V \lhd [(\mathsf{Some}\,l', v')] \sqcup V'$
et $\tau \lhd [\mathsf{Tag}\ \psi\ \mathsf{IMPLICIT}]$.
Distinguons des hypothèses de filtre :
— **HF.0** : $\tau \lhd [\mathsf{Tag}\ \psi\ \mathsf{IMPLICIT}]$
— **HF.1** : $\mathrm{T} \lhd \mathsf{SET}\,(\varphi' :: \Phi')$
Les autres hypothèses sont
— **H.1** : $\langle\mathcal{A}\rangle\Gamma, x \vdash_{\mathcal{C}} \tau\,\mathrm{T}$. Remarquons que $\Phi \lhd \varphi' :: \Phi'$ implique $\Phi \lhd [\varphi'] \sqcup \Phi'$, où φ' et Φ' sont identiques dans les deux jugements. Par conséquent, en tenant compte des hypothèses ci-dessus, nous pouvons envisager l'application de la règle [22] page 168 :

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s')\\ \langle\mathcal{A}\rangle\Gamma, \text{""} \vdash_{\mathcal{C}} \tau'\,\mathrm{T}' \qquad \langle\mathcal{A}\rangle\Gamma, x \vdash_{\mathcal{C}} \tau\ (\mathsf{SET}\,\Phi')\end{array}}{\langle\mathcal{A}\rangle\Gamma, x \vdash_{\mathcal{C}} \tau\,(\mathsf{SET}\,([\varphi'] \sqcup \Phi'))}\ [22]$$

Appliquons directement cette règle (i.e. toutes les variables de la règle sont liées aux variables de même nom dans l'environnement du cas courant de la preuve). Étiquetons ses prémisses non redondantes avec les hypothèses de filtre :
— **H.1.0** : $\langle\mathcal{A}\rangle\Gamma, x \vdash_{\mathcal{C}} \tau'\,\mathrm{T}'$
— **H.1.1** : $\langle\mathcal{A}\rangle\Gamma, x \vdash_{\mathcal{C}} \tau\ (\mathsf{SET}\,\Phi')$

— **H.2** : $\langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathrm{T}$. La règle [18] page 148 peut s'appliquer :

$$\frac{\langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathsf{SEQUENCE}\,\Phi}{\langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathsf{SET}\,\Phi}\ [18]$$

Appliquons-là directement et étiquetons sa premisse :
— **H.2.0** : $\langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathsf{SEQUENCE}\,\Phi$
Remarquons que $\Phi \lhd \varphi' :: \Phi'$ implique $\Phi \lhd [\varphi'] \sqcup \Phi'$, où φ' et Φ' sont identiques dans les deux jugements. Par conséquent, en tenant compte des hypothèses ci-dessus, nous pouvons envisager l'application de la règle [15] page 147 :

$$\frac{\begin{array}{cc}\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') & V \lhd [(\mathsf{Some}\,l', v')] \sqcup V'\\ \langle\mathcal{A}\rangle\Gamma, \text{""} \vdash v' : \mathrm{T}' \qquad \langle\mathcal{A}\rangle\Gamma, x \vdash \{V'\} : \mathsf{SEQUENCE}\,\Phi' & \end{array}}{\langle\mathcal{A}\rangle\Gamma, x \vdash \{V\} : \mathsf{SEQUENCE}\,([\varphi'] \sqcup \Phi')}\ [15]$$

Appliquons directement cette règle et étiquetons ses prémisses non redondantes avec les hypothèses de filtre :
— **H.2.0.0** : $\langle\mathcal{A}\rangle\Gamma, "" \vdash v' : \mathrm{T}'$
— **H.2.0.1** : $\langle\mathcal{A}\rangle\Gamma, x \vdash \{V'\} : \mathsf{SEQUENCE}\,\Phi'$
H.2.0.1 est une instance possible pour l'unique prémisse de la règle [18] page 148 :

$$\frac{\langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathsf{SEQUENCE}\,\Phi}{\langle\mathcal{A}\rangle\Gamma, x \vdash v : \mathsf{SET}\,\Phi}\ [18]$$

qui permet alors de dériver directement
— **H.9** : $\langle\mathcal{A}\rangle\Gamma, x \vdash \{V'\} : \mathsf{SET}\,\Phi'$
— **H.3** : $\langle\mathcal{A}\rangle\Gamma \vdash v : \tau\,\mathrm{T} \Rightarrow \tilde{v}$. La règle [21] page 161 peut s'appliquer :

$$\frac{\begin{array}{c}\langle\mathcal{A}\rangle\Gamma \vdash v : \tau\,(\mathsf{SEQUENCE}\,\Phi) \Rightarrow (\psi, \mathsf{Const}\,\tilde{V}) \\ \pi\ \textit{est une permutation sur}\ \tilde{V}\end{array}}{\langle\mathcal{A}\rangle\Gamma \vdash v : \tau\,(\mathsf{SET}\,\Phi) \Rightarrow (\psi, \mathsf{Const}\,(\pi\ (\tilde{V})))}\ [21]$$

Pour éviter une capture *a priori* du nom ψ (lié dans l'environnement courant du cas de la preuve par **HF.0**) lors de l'application, effectuons tout de suite le renommage suivant :
— $\psi \xleftarrow{\alpha} \overline{\psi}$
et appliquons alors la nouvelle règle directement. (Insistons sur le fait qu'il ne s'agissait pas d'une substitution de ψ, car la variable ψ n'est pas liée dans un champ projeté de la conclusion.) Étiquetons alors ses prémisses :
— **H.3.0** : $\langle\mathcal{A}\rangle\Gamma \vdash v : \tau\,(\mathsf{SEQUENCE}\,\Phi) \Rightarrow (\overline{\psi}, \mathsf{Const}\,\tilde{V})$
— **H.3.1** : π *est une permutation sur* $\tilde{V}$
et aussi le jugement résultant de la projection du champ inféré :
— **H.3.2** : $(\overline{\psi}, \mathsf{Const}\,(\pi\ (\tilde{V}))) \lhd \tilde{v}$
Il est d'autre part aisé de démontrer par induction le théorème suivant (considérer les règles [18] page 160, [19] page 161 et [20] page 161 du codage) :

$$\begin{array}{ll} \textit{Soit} & \mathrm{T} \lhd \mathsf{SEQUENCE}\,\Phi \\ \textit{et} & \tau \lhd [\mathsf{Tag}\ \psi\ \mathsf{IMPLICIT}] \\ \textit{Si} & \langle\mathcal{A}\rangle\Gamma \vdash v : \tau\,\mathrm{T} \Rightarrow (\overline{\psi}, \mathsf{Const}\,\tilde{V}) \\ \textit{alors} & \overline{\psi} = \psi \end{array}$$

Il découle de l'application de ce théorème :
— **H.4** : $\overline{\psi} = \psi$

Les propositions **H.3.0** et **H.4** impliquent :
— **H.10** : $\langle\mathcal{A}\rangle\Gamma \vdash v : \tau\,(\mathsf{SEQUENCE}\,\Phi) \Rightarrow (\psi, \mathsf{Const}\,\tilde{V})$
Les propositions **H.3.2** et **H.4** impliquent :
— **H.11** : $(\psi, \mathsf{Const}\,(\pi\ (\tilde{V}))) \lhd \tilde{v}$
En tenant compte des hypothèses ci-dessus, nous pouvons envisager l'application de la règle [18] page 160, à partir du jugement **H.10** :

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \\ V \lhd [(\mathsf{Some}\,l', v')] \sqcup V' \qquad \langle\mathcal{A}\rangle\Gamma \vdash v' : \tau'\,\mathrm{T}' \Rightarrow \tilde{v}' \\ \langle\mathcal{A}\rangle\Gamma \vdash \mathbf{\{}V'\mathbf{\}} : \tau\,(\mathsf{SEQUENCE}\,\Phi') \Rightarrow (\psi, \mathsf{Const}\,\tilde{V}') \\ \tilde{v} \triangleq (\psi, \mathsf{Const}\,(\tilde{v}' :: \tilde{V}'))\end{array}}{\langle\mathcal{A}\rangle\Gamma \vdash \mathbf{\{}V\mathbf{\}} : \tau\,(\mathsf{SEQUENCE}\,(\varphi' :: \Phi')) \Rightarrow \tilde{v}}\ [18]$$

Pour éviter une capture du nom $\tilde{v}$ qui est lié dans l'environnement du cas courant de la preuve par **H.11**, effectuons d'abord le renommage :
— $\tilde{v} \xleftarrow{\alpha} \tilde{v}_0$
Appliquons cette règle, et étiquetons alors ses prémisses non redondantes avec les hypothèses de filtre.
— **H.10.0** : $\langle\mathcal{A}\rangle\Gamma \vdash v' : \tau'\,\mathrm{T}' \Rightarrow \tilde{v}'$
— **H.10.1** : $\langle\mathcal{A}\rangle\Gamma \vdash \mathbf{\{}V'\mathbf{\}} : \tau\,(\mathsf{SEQUENCE}\,\Phi') \Rightarrow (\psi, \mathsf{Const}\,\tilde{V}')$
— **H.10.3** : $\tilde{v}_0 \triangleq (\psi, \mathsf{Const}\,(\tilde{v}' :: \tilde{V}'))$
Ici nous devons identifier le terme $\tilde{v}_0$ et le champ inféré du jugement **H.10** parce que, dans le jugement **H.10**, à partir duquel nous appliquons cette règle, $(\psi, \mathsf{Const}\,\tilde{V})$ est une valeur, et non un motif. Soit :
— **H.12** : $\tilde{v}_0 = (\psi, \mathsf{Const}\,\tilde{V})$
Les propositions **H.10.3** et **H.12** impliquent :
— **H.13** : $\tilde{v}' :: \tilde{V}' = \tilde{V}$

La proposition **H.10.1** est une instance possible pour la première prémisse de la règle [21] page 161 :

$$\frac{\begin{array}{c}\langle\mathcal{A}\rangle\Gamma \vdash v : \tau\,(\mathsf{SEQUENCE}\,\Phi) \Rightarrow (\psi, \mathsf{Const}\,\tilde{V}) \\ \pi \textit{ est une permutation sur } \tilde{V}\end{array}}{\langle\mathcal{A}\rangle\Gamma \vdash v : \tau\,(\mathsf{SET}\,\Phi) \Rightarrow (\psi, \mathsf{Const}\,(\pi\ (\tilde{V})))}\ [21]$$

Pour éviter une capture *a priori* du nom π (lié dans l'environnement courant du cas de la preuve par **H.3.1**) lors de la dérivation, effectuons tout de suite le renommage suivant :
— $\pi \xleftarrow{\alpha} \overline{\pi}$
Dérivons un jugement à l'aide de cette règle, après ce renommage. Ce faisant nous effectuons les substitutions suivantes :

— $\Phi \xleftarrow{\beta} \Phi'$
— $\tilde{V} \xleftarrow{\beta} \tilde{V}'$
— $v \xleftarrow{\beta} \{V'\}$
La conclusion est alors :
— **H.14** : $\langle\mathcal{A}\rangle\Gamma \vdash \{V'\} : \tau\,(\mathsf{SET}\,\Phi') \Rightarrow (\psi, \mathsf{Const}\,(\overline{\pi}\ (\tilde{V}')))$
avec
— **H.14.0** : $\overline{\pi}$ *est une permutation sur* $\tilde{V}'$
Maintenant

1. Prouvons **Hyp** $\Rightarrow$ **PA**. On se donne :
   — **PA.H** : $\vdash_{\mathcal{D}} \tilde{v} \to e$. La propositiion **HF.0** implique
   — is_built_in_type (ψ) = **true**
   Donc, en tenant compte de **H.11**, la règle [2] page 164 s'applique :

$$\frac{\mathrm{is\_built\_in\_tag}\,(\psi) \qquad \mathrm{T} \neq \mathsf{TRef}\ \ \_ \mid \_ < \_\_ \mid \mathsf{CHOICE}\ \ \_}{\vdash_{\mathcal{D}} (\psi, \mathsf{Const}\_) \to ([\mathsf{Tag}\ \psi\ \mathsf{IMPLICIT}], \mathrm{T})}\ [2]$$

   Appliquons directement cette règle, après le renommage
   — $\mathrm{T} \xleftarrow{\alpha} \mathrm{T}_0$
   Étiquetons ses prémisses non redondantes :
   — **PA.H.0** : $\mathrm{T}_0 \neq \mathsf{TRef}\ \ \_ \mid \_ < \_\_ \mid \mathsf{CHOICE}\ \ \_$
   — **PA.H.1** : $e \triangleq ([\mathsf{Tag}\ \psi\ \mathsf{IMPLICIT}], \mathrm{T}_0)$
   Notons que la proposition **PA.H.1** définit la liaison de nom e (la variable e n'est pas substituée lors de l'application de la règle car elle lie une inférence).
   Prouvons alors :
   — **PA.C** : $\langle\mathcal{A}\rangle\Gamma \vdash e \sqsubseteq (\tau, \mathrm{T})$. Les propositions **HF.0**, **HF.1**, **PA.H.0** et **PA.H.1** impliquent que c'est la règle [4] page 161 qui s'applique :

$$\frac{\langle\mathcal{A}\rangle\Gamma \vdash (\tau'_0, \mathrm{T}_0) \sqsubseteq (\tau'_1, \mathrm{T}_1)}{\langle\mathcal{A}\rangle\Gamma \vdash (t :: \tau'_0, \mathrm{T}_0) \sqsubseteq (t :: \tau'_1, \mathrm{T}_1)}\ [4]$$

   Appliquons-là. Ce faisant nous effectuons sur la règle les substitutions suivantes :
   — $\mathrm{T}_1 \xleftarrow{\beta} \mathrm{T}$
   — $t \xleftarrow{\beta} \mathsf{Tag}\ \psi\ \mathsf{IMPLICIT}$
   — $\tau_0 \xleftarrow{\beta} [\,]$
   — $\tau_1 \xleftarrow{\beta} [\,]$
   Puis étiquetons alors sa prémisse :
   — **PA.C.0** : $\langle\mathcal{A}\rangle\Gamma \vdash ([\,], \mathrm{T}_0) \sqsubseteq ([\,], \mathrm{T})$

Or la proposition **PA.C.0** peut être dérivée par la règle [8] page 161 de la relation d'inclusion d'étiquetage :

$$\frac{\begin{array}{l}\mathrm{T}_0 \not\lhd \mathsf{CHOICE}\ \ \_ \mid \mathsf{TRef}\ \ \_ \mid \_ < \_ \_ \\ \mathrm{T}_1 \not\lhd \mathsf{CHOICE}\ \ \_ \mid \mathsf{TRef}\ \ \_ \mid \_ < \_ \_\end{array}}{\langle\mathcal{A}\rangle\Gamma \vdash ([\,], \mathrm{T}_0) \sqsubseteq ([\,], \mathrm{T}_1)}\ [8]$$

Ce faisant nous effectuons la substitution suivante :
— $\mathrm{T}_1 \xleftarrow{\beta} \mathrm{T}$
et nous reconnaissons que les propositions **PA.H.0** et **HF.1** permettent d'instancier les premisses. La proposition **PA.C.0** implique par définition **PA.C**. ◊

2. Prouvons **Hyp** ⇒ **C**.
— $\mathbf{C} : \langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\,\mathrm{T}$
La proposition **H.12** nous permet d'envisager l'application de la règle [19] page 166 :

$$\frac{\begin{array}{c}\mathrm{T} \lhd \mathsf{SET}\,([\varphi'] \sqcup \Phi') \qquad \varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \\ \tilde{V} \lhd [\tilde{v}'] \sqcup \tilde{V}' \qquad \langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}(\tilde{v}') \sqsubseteq (\tau', \mathrm{T}') \\ \langle\mathcal{A}\rangle\Gamma \models \tilde{v}' : \tau'\,\mathrm{T}' \qquad \langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Const}\,\tilde{V}') : \tau\ (\mathsf{SET}\,\Phi')\end{array}}{\langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Const}\,\tilde{V}) : \tau\,\mathrm{T}}\ [19]$$

Pour éviter *a priori* une capture des noms $\tilde{v}'$ et $\tilde{V}'$, Effectuons d'abord les renommages suivants :
— $\tilde{v}' \xleftarrow{\alpha} \tilde{v}'_0$
— $\tilde{V}' \xleftarrow{\alpha} \tilde{V}'_0$
Appliquons ensuite la règle, en effectuant la substitution suivante (à cause de **H.12**) :
— $\tilde{V} \xleftarrow{\beta} \pi\,(\tilde{V})$
Remarquons que $\Phi \lhd \varphi' :: \Phi'$ implique $\Phi \lhd [\varphi'] \sqcup \Phi'$, où φ' et Φ' sont identiques dans les deux jugements. Étiquetons les prémisses non redondantes avec les hypothèses de filtre :
— $\mathbf{C.0} : \pi\,(\tilde{V}) \lhd [\tilde{v}'_0] \sqcup \tilde{V}'_0$
— $\mathbf{C.1} : \langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}(\tilde{v}'_0) \sqsubseteq (\tau', \mathrm{T}')$
— $\mathbf{C.2} : \langle\mathcal{A}\rangle\Gamma \models \tilde{v}'_0 : \tau'\,\mathrm{T}'$
— $\mathbf{C.3} : \langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Const}\,\tilde{V}'_0) : \tau\ (\mathsf{SET}\,\Phi')$
Par ailleurs, π et $\overline{\pi}$ sont des méta-variables dont les uniques prédicats sont **H.3.1** et **H.10.1.1**, et nous avons **H.13**. Nous pouvons alors choisir π et $\overline{\pi}$ tels que :
— $\pi\,(\tilde{V}) = [\tilde{v}'] \sqcup \overline{\pi}\,(\tilde{V}')$
Donc si l'on suppose :
— $\mathbf{H.15} : \tilde{v}'_0 = \tilde{v}'$

— **H.16** : $\tilde{V}'_0 = \overline{\pi}\,(\tilde{V}')$
alors **C.0** est vraie et il nous reste à démontrer **C.1**, **C.2** et **C.3**. Dans un premier temps reprenons l'énoncé du théorème et effectuons-y les renommages suivants :
— $v \xleftarrow{\alpha} v'$
— $\tau \xleftarrow{\alpha} \tau'$
— $\mathrm{T} \xleftarrow{\alpha} \mathrm{T}'$
— $\tilde{v} \xleftarrow{\alpha} \tilde{v}'$
Nous obtenons alors :

$$\begin{array}{ll} \mathit{Soit} & \langle\mathcal{A}\rangle\Gamma, x \vdash_{\overline{c}} \tau'\,\mathrm{T}' \\ \mathit{Si} & \langle\mathcal{A}\rangle\Gamma, x \vdash v' : \mathrm{T}' \\ \mathit{et} & \langle\mathcal{A}\rangle\Gamma \vdash v' : \tau'\,\mathrm{T}' \Rightarrow \tilde{v}' \\ \mathit{alors} & \langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}') \sqsubseteq (\tau', \mathrm{T}') \\ \mathit{et} & \langle\mathcal{A}\rangle\Gamma \models \tilde{v}' : \tau'\,\mathrm{T}' \end{array}$$

Nous reconnaissons en cet énoncé une instance de l'hypothèse d'induction, car ses prémisses sont, dans l'ordre d'écriture, **H.1.0**, **H.2.0.0** et **H.10.0**. En l'appliquant il vient
— **H.5** : $\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}') \sqsubseteq (\tau', \mathrm{T}')$
— **H.6** : $\langle\mathcal{A}\rangle\Gamma \models \tilde{v}' : \tau'\ \mathrm{T}'$
Dans un deuxième temps, reprenons l'énoncé du théorème et effectuons-y les renommages suivants :
— $\mathrm{T} \xleftarrow{\alpha} \mathsf{SET}\,\Phi'$
— $\tilde{v} \xleftarrow{\alpha} (\psi, \mathsf{Const}\,(\overline{\pi}\,(\tilde{V}')))$
— $v \xleftarrow{\alpha} \{V'\}$
Nous obtenons alors :

$$\begin{array}{ll} \mathit{Soit} & \langle\mathcal{A}\rangle\Gamma, x \vdash_{\overline{c}} \tau\,(\mathsf{SET}\,\Phi') \\ \mathit{Si} & \langle\mathcal{A}\rangle\Gamma, x \vdash \{V'\} : \mathsf{SET}\,\Phi' \\ \mathit{et} & \langle\mathcal{A}\rangle\Gamma \vdash \{V'\} : \tau\,(\mathsf{SET}\,\Phi') \Rightarrow (\psi, \mathsf{Const}\,(\overline{\pi}\,(\tilde{V}'))) \\ \mathit{alors} & \langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\psi, \mathsf{Const}\,(\overline{\pi}\,(\tilde{V}'))) \sqsubseteq (\tau, \mathsf{SET}\,\Phi') \\ \mathit{et} & \langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Const}\,(\overline{\pi}\,(\tilde{V}'))) : \tau\,(\mathsf{SET}\,\Phi') \end{array}$$

Nous reconnaissons en cet énoncé une instance de l'hypothèse d'induction, car ses prémisses sont, dans l'ordre d'écriture, **H.1.1**, **H.9** et **H.14**. En l'appliquant il vient :
— **H.7** : $\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\psi, \mathsf{Const}\,(\overline{\pi}\,(\tilde{V}'))) \sqsubseteq (\tau, \mathsf{SET}\,\Phi')$
— **H.8** : $\langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Const}\,(\overline{\pi}\,(\tilde{V}'))) : \tau\ (\mathsf{SET}\,\Phi')$
On a :
— **H.5** et **H.15** impliquent **C.1**.
— **H.6** et **H.15** impliquent **C.2**.
— **H.8** et **H.16** impliquent **C.3**.

Les propositions **C.0**, **C.1**, **C.2** et **C.3** impliquent par définition **C**. ◊

Conclusion : $\left.\begin{array}{l}\mathbf{Hyp} \Rightarrow \mathbf{PA} \\ \mathbf{Hyp} \Rightarrow \mathbf{C}\end{array}\right\}$ impliquent $\mathbf{Hyp} \Rightarrow (\mathbf{PA} \wedge \mathbf{C})$. △

On note que **H.7** est inutile.

— $\mathrm{T} \lhd \mathsf{SET}\,([\varphi'] \sqcup \Phi')$ et $\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s')$
et $s' \lhd \mathsf{Some}\,\mathsf{OPTIONAL}$
et $\tau \lhd [\mathsf{Tag}\ \psi\ \mathsf{IMPLICIT}]$.
Cas laissé au lecteur.

□

8.5 Unicité du codage

Théorème 8.5.1 (Unicité du codage)**.**

S'il existe un codage d'une valeur d'un type bien labellisé, alors ce codage est unique.

Preuve du théorème 8.5.1

La preuve sera la même que celle concernant l'unicité du contrôle de type (7.3.1 page 149) sauf que l'on utilisera respectivement les lemmes 8.5.2, 8.5.3 page suivante, en lieu des lemmes 7.3.4 page 153 et 7.3.3 page 152.

Remarque

Le cas du type SET est identique du type SEQUENCE, la seule difference étant dans le réordonnancement éventuel des sous-parties du code :

$$\frac{\begin{array}{c}\langle\mathcal{A}\rangle\Gamma \vdash v : \tau\,(\mathsf{SEQUENCE}\,\Phi) \Rightarrow (\psi, \mathsf{Const}\,\tilde{V}) \\ \pi \textit{ est une permutation sur } \tilde{V}\end{array}}{\langle\mathcal{A}\rangle\Gamma \vdash v : \tau\,(\mathsf{SET}\,\Phi) \Rightarrow (\psi, \mathsf{Const}\,(\pi\ (\tilde{V})))}\ [21]$$

Lemme 8.5.2.

$$\begin{array}{ll} \textit{Si} & \langle\mathcal{A}\rangle\Gamma, \{x'\} \uplus L \vdash_{\mathcal{L}} \mathit{CHOICE}\,\mathcal{F} \\ \textit{alors} & \neg(\langle\mathcal{A}\rangle\Gamma \vdash (x' : v') : \tau\ (\mathit{CHOICE}\,\mathcal{F}) \Rightarrow \tilde{v}) \end{array}$$

Preuve du lemme 8.5.2

La preuve est la même que celle du lemme 7.3.4 page 153, mais avec l'algorithme de codage au lieu du contrôle des types. Cependant les règles [13] page 146) et [14] page 146 deviennent respectivement [16] page 160 et [17]) (p. 160). □

Lemme 8.5.3.

$$\begin{array}{ll} Soit & v \lhd \{[(\mathit{Some}\, x', v')] \sqcup V'\} \\ Si & \langle\mathcal{A}\rangle\Gamma, \{x'\} \uplus L \vdash_{\mathcal{L}} \mathit{SEQUENCE}\, \Phi \\ alors & \neg(\langle\mathcal{A}\rangle\Gamma \vdash v : \mathit{SEQUENCE}\, \Phi \Rightarrow \tilde{v}) \end{array}$$

Preuve du théorème 8.5.3

La preuve est la même que celle du théorème 7.3.3 page 152, mais avec l'algorithme de codage au lieu du contrôle de type. Cependant les règles [15] page 147, [16] page 147 et [17] page 147 deviennent respectivement [18] page 160, [19] page 161 et [20] page 161. □

8.6 Types bien étiquetés

$$\frac{\langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{T}} \mathrm{T}}{\langle\mathcal{A}\rangle\Gamma \vdash_{\mathcal{T}} \mathrm{T}}\ [1]$$

$$\frac{x' \in \mathcal{A} \qquad \Gamma(x') \lhd (\alpha', \tau', \{(\mathrm{T}', \sigma')\}) \qquad \langle\mathcal{A}\rangle\Gamma, \mathrm{E} \vdash_{\mathcal{T}} \mathrm{T}'}{\langle\mathcal{A}\rangle\Gamma, \mathrm{E} \vdash_{\mathcal{T}} \mathsf{TRef}\,(x')}\ [2]$$

$$\frac{\begin{array}{c} f' \lhd (l', \tau', \mathrm{T}', \sigma', s') \qquad \langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{T}} \mathrm{T}' \\ \forall e \in \mathrm{E}.\neg(\langle\mathcal{A}\rangle\Gamma \vdash e \parallel (\tau', \mathrm{T}')) \qquad \langle\mathcal{A}\rangle\Gamma, \{(\tau', \mathrm{T}')\} \cup \mathrm{E} \vdash_{\mathcal{T}} \mathsf{CHOICE}\, \mathcal{F}' \end{array}}{\langle\mathcal{A}\rangle\Gamma, \mathrm{E} \vdash_{\mathcal{T}} \mathsf{CHOICE}\,([f'] \sqcup \mathcal{F}')}\ [3]$$

$$\frac{\begin{array}{c} \varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \qquad \langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{T}} \mathrm{T}' \\ \forall e \in \mathrm{E}.\neg(\langle\mathcal{A}\rangle\Gamma \vdash e \parallel (\tau', \mathrm{T}')) \qquad \langle\mathcal{A}\rangle\Gamma, \{(\tau', \mathrm{T}')\} \cup \mathrm{E} \vdash_{\mathcal{T}} \mathsf{SET}\, \Phi' \end{array}}{\langle\mathcal{A}\rangle\Gamma, \mathrm{E} \vdash_{\mathcal{T}} \mathsf{SET}\,([\varphi'] \sqcup \Phi')}\ [4]$$

$$\frac{\begin{array}{c} \varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', \mathsf{None}) \qquad \langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{T}} \mathrm{T}' \\ \forall e \in \mathrm{E}.\neg(\langle\mathcal{A}\rangle\Gamma \vdash e \parallel (\tau', \mathrm{T}')) \qquad \langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{T}} \mathsf{SEQUENCE}\, \Phi' \end{array}}{\langle\mathcal{A}\rangle\Gamma, \mathrm{E} \vdash_{\mathcal{T}} \mathsf{SEQUENCE}\,(\varphi' :: \Phi')}\ [5]$$

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \\ s' \not\lhd \mathsf{None} \qquad \langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{T}} \mathrm{T}' \qquad \forall e \in \mathrm{E}.\neg(\langle\mathcal{A}\rangle\Gamma \vdash e \parallel (\tau', \mathrm{T}')) \\ \langle\mathcal{A}\rangle\Gamma, \{(\tau', \mathrm{T}')\} \cup \mathrm{E} \vdash_{\mathcal{T}} \mathsf{SEQUENCE}\,\Phi'\end{array}}{\langle\mathcal{A}\rangle\Gamma, \mathrm{E} \vdash_{\mathcal{T}} \mathsf{SEQUENCE}\,(\varphi' :: \Phi')}\ [6]$$

$$\frac{\langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{T}} \mathrm{T}'}{\langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{T}} (\mathsf{SEQUENCE\ OF} \mid \mathsf{SET\ OF})\,\tau'\,\mathrm{T}'\,\sigma'}\ [7]$$

$$\frac{\begin{array}{c}\mathrm{T} \not\lhd \mathsf{TRef}\ \_ \mid \mathsf{CHOICE}\ \_ \\ \mid \mathsf{SET\ OF}\ \_\_\_ \mid \mathsf{SEQUENCE\ OF}\ \_\_\_ \\ \mid \mathsf{SET}\ \_ \mid \mathsf{SEQUENCE}\ \_\end{array}}{\langle\mathcal{A}\rangle\Gamma, \mathrm{E} \vdash_{\mathcal{T}} \mathrm{T}}\ [8]$$

8.7 Unicité du contrôle sémantique des types

Théorème 8.7.1 (Unicité du contrôle de type sémantique)**.**

S'il existe une preuve qu'une valeur codée est d'un type bien étiqueté, alors cette preuve est unique.

Nous avons besoin de deux lemmes techniques au préalable.

Théorème 8.7.2.

Si	$\langle\mathcal{A}\rangle\Gamma, \{e'\} \cup E \vdash_{\mathcal{T}} \mathsf{CHOICE}\,\mathcal{F}$
et	$\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}) \sqsubseteq e'$
alors	$\neg(\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{CHOICE}\,\mathcal{F}))$

Preuve du lemme 8.7.2

La démonstration se fera par induction.
— $\mathcal{F} \lhd [f'] \sqcup \mathcal{F}'$ et $f' \lhd (l', \tau', \mathrm{T}', \sigma', s')$ et $v \lhd x' : v'$

Les autres hypothèses sont

— **H.0** : $\langle\mathcal{A}\rangle\Gamma, \{e'\} \cup \mathrm{E} \vDash_{\mathcal{T}} \mathsf{CHOICE}\,\mathcal{F}$. C'est la règle [3] page 197 qui s'applique :

$$\frac{\begin{array}{c} f' \lhd (l', \tau', \mathrm{T}', \sigma', s') \\ \langle\mathcal{A}\rangle\Gamma, \{\} \vDash_{\mathcal{T}} \mathrm{T}' \qquad \forall e \in \mathrm{E}.\neg(\langle\mathcal{A}\rangle\Gamma \vdash e \parallel (\tau', \mathrm{T}')) \\ \langle\mathcal{A}\rangle\Gamma, \{(\tau', \mathrm{T}')\} \cup \mathrm{E} \vDash_{\mathcal{T}} \mathsf{CHOICE}\,\mathcal{F}' \end{array}}{\langle\mathcal{A}\rangle\Gamma, \mathrm{E} \vDash_{\mathcal{T}} \mathsf{CHOICE}\,([f'] \sqcup \mathcal{F}')}\ [3]$$

Appliquons-là. Ce faisant nous effectuons la substitution :
— $\mathrm{E} \xleftarrow{\beta} \{e'\} \cup \mathrm{E}$
Étiquetons ensuite ses prémisses non redondantes avec les hypothèses de filtre :
— **H.0.0** : $\langle\mathcal{A}\rangle\Gamma, \{\} \vDash_{\mathcal{T}} \mathrm{T}'$
— **H.0.1** : $\forall e \in \{e'\} \cup \mathrm{E}.\neg(\langle\mathcal{A}\rangle\Gamma \vdash e \parallel (\tau', \mathrm{T}'))$
— **H.0.2** : $\langle\mathcal{A}\rangle\Gamma, \{(\tau', \mathrm{T}')\} \cup (\{e'\} \cup \mathrm{E}) \vDash_{\mathcal{T}} \mathsf{CHOICE}\,\mathcal{F}'$
— **H.1** : $\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}) \sqsubseteq e'$
Prouvons
— **C** : $\neg(\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{CHOICE}\,\mathcal{F}))$
Les seules règles du contrôle sémantique des types dérivant des jugements sur les types CHOICE sont :

$$\frac{\begin{array}{c} f' \lhd (l', \tau', \mathrm{T}', \sigma', s') \\ \langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}(\tilde{v}) \sqsubseteq (\tau', \mathrm{T}') \qquad \langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau'\,\mathrm{T}' \end{array}}{\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : [\,]\ (\mathsf{CHOICE}\,([f'] \sqcup \mathcal{F}'))}\ [14]$$

$$\frac{\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{CHOICE}\,\mathcal{F}')}{\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{CHOICE}\,([f'] \sqcup \mathcal{F}'))}\ [15]$$

La proposition **H.0.1** implique
— **H.2** : $\neg(\langle\mathcal{A}\rangle\Gamma \vdash e' \parallel (\tau', \mathrm{T}'))$
La propriété de symétrie de $\parallel$ (Prop. 8.2.2.3 page 162) et **H.2** impliquent :
— **H.3** : $\neg(\langle\mathcal{A}\rangle\Gamma \vdash (\tau', \mathrm{T}') \parallel e')$
Enfin, **H.1** et **H.3** et le théorème 8.2.2.4 page 163 impliquent
— **H.4** : $\neg(\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}(\tilde{v}') \sqsubseteq (\tau', \mathrm{T}'))$
Or **H.4** est la négation de la première prémisse de la règle [14], qui ne peut donc s'appliquer. Il ne nous reste alors qu'à prouver que la règle [15] ne peut s'appliquer, pour prouver le théorème. Reprenons l'énoncé du théorème et effectuons-y les renommages suivants :
— $\mathrm{E} \xleftarrow{\alpha} \{(\tau', \mathrm{T}')\} \cup \mathrm{E}$
— $\mathcal{F} \xleftarrow{\alpha} \mathcal{F}'$

Nous obtenons alors :

$$\begin{array}{ll} Si & \langle\mathcal{A}\rangle\Gamma, \{e'\} \cup (\{(\tau', \mathrm{T}')\} \cup \mathrm{E}) \vdash_{\mathcal{T}} \mathsf{CHOICE}\,\mathcal{F}' \\ et & \langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}) \sqsubseteq e' \\ alors & \neg(\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{CHOICE}\,\mathcal{F}')) \end{array}$$

Nous reconnaissons en cet énoncé une instance de l'hypothèse d'induction, car ses prémisses sont, dans l'ordre d'écriture, **H.0.2** et **H.1**. En l'appliquant il vient donc :
— **H.5** : $\neg(\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{CHOICE}\,\mathcal{F}'))$
Or **H.5** est la négation de la prémisse de la règle [15], qui ne peut donc être appliquée. Puisque les deux seules règles pouvant dériver

$$\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{CHOICE}\,\mathcal{F})$$

ne peuvent être appliquées, cela signifie que **C** est vraie. △

— $\mathcal{F} \lhd [\,]$ et $v \lhd x' : v'$

Les autres hypothèses sont
— **H.0** : $\langle\mathcal{A}\rangle\Gamma, \{e'\} \cup \mathrm{E} \vdash_{\mathcal{T}} \mathsf{CHOICE}\,\mathcal{F}$
— **H.1** : $\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}) \sqsubseteq e'$
Prouvons
— **C** : $\neg(\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{CHOICE}\,\mathcal{F}))$
Ni la règle [14] page 165, ni la règle [15] page 165 ne peuvent être appliquées car $\mathcal{F} \lhd [\,]$ (aucun des deux filtres n'accepte cette valeur). Puisque les deux seules règles pouvant dériver $\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{CHOICE}\,\mathcal{F})$ ne peuvent être appliquées, cela signifie que **C** est vraie. △

□

Théorème 8.7.3.

$$\begin{array}{ll} Soit & \tilde{v} \lhd (\psi, \mathit{Const}\,(\tilde{v}' :: \tilde{V}')) \\ Si & \langle\mathcal{A}\rangle\Gamma, \{e'\} \cup E \vdash_{\mathcal{T}} \mathit{SEQUENCE}\,\Phi \\ et & \langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}') \sqsubseteq e' \\ alors & \neg(\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathit{SEQUENCE}\,\Phi)) \end{array}$$

Preuve du lemme 8.7.3

La démonstration se fera par induction.

— $\Phi \lhd \varphi' :: \Phi'$ et $\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s')$
et $\tilde{v} \lhd (\psi, \mathsf{Const}\,(\tilde{v}' :: \tilde{V}'))$ et $s' \lhd \mathsf{None}$

Distinguons des hypothèses de filtre :
— **HF.0** : $s' \lhd \mathsf{None}$
— **HF.1** : $\Phi \lhd \varphi' :: \Phi'$
Les autres hypothèses sont
— **H.0** : $\langle \mathcal{A} \rangle \Gamma, \{e'\} \cup \mathrm{E} \vdash_{\mathcal{T}} \mathsf{SEQUENCE}\,\Phi$. Les propositions **HF.0** et **HF.1** font que la règle [5] page 197 s'applique :

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', \mathsf{None}) \\ \langle \mathcal{A} \rangle \Gamma, \{\} \vdash_{\mathcal{T}} \mathrm{T}' \qquad \forall e \in \mathrm{E}. \neg(\langle \mathcal{A} \rangle \Gamma \vdash e \parallel (\tau', \mathrm{T}')) \\ \langle \mathcal{A} \rangle \Gamma, \{\} \vdash_{\mathcal{T}} \mathsf{SEQUENCE}\,\Phi'\end{array}}{\langle \mathcal{A} \rangle \Gamma, \mathrm{E} \vdash_{\mathcal{T}} \mathsf{SEQUENCE}\,(\varphi' :: \Phi')}\ [5]$$

Appliquons-là. Ce faisant nous effectuons la substitution
— $\mathrm{E} \xleftarrow{\beta} \{e'\} \cup \mathrm{E}$
Étiquetons ses prémisses :
— **H.0.0** : $\langle \mathcal{A} \rangle \Gamma, \{\} \vdash_{\mathcal{T}} \mathrm{T}'$
— **H.0.1** : $\forall e \in \{e'\} \cup \mathrm{E}. \neg(\langle \mathcal{A} \rangle \Gamma \vdash e \parallel (\tau', \mathrm{T}'))$
— **H.0.2** : $\langle \mathcal{A} \rangle \Gamma, \{\} \vdash_{\mathcal{T}} \mathsf{SEQUENCE}\,\Phi'$
— **H.1** : $\langle \mathcal{A} \rangle \Gamma \vdash \mathcal{D}\,(\tilde{v}') \sqsubseteq e'$
Prouvons
— **C** : $\neg(\langle \mathcal{A} \rangle \Gamma \models \tilde{v} : \tau\ (\mathsf{SEQUENCE}\,\Phi))$
Les seules règles dérivant des jugements sur les types SEQUENCE sont celles numérotées [16] page 165, [17] page 165 et [18] page 165 :

$$\frac{\begin{array}{c}\mathrm{T} \lhd \mathsf{SEQUENCE}\,(\varphi' :: \Phi') \qquad \varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \\ \tilde{V} \lhd \tilde{v}' :: \tilde{V}' \qquad \langle \mathcal{A} \rangle \Gamma \vdash \mathcal{D}(\tilde{v}') \sqsubseteq (\tau', \mathrm{T}') \qquad \langle \mathcal{A} \rangle \Gamma \models \tilde{v}' : \tau'\,\mathrm{T}' \\ \langle \mathcal{A} \rangle \Gamma \models (\psi, \mathsf{Const}\,\tilde{V}') : \tau\ (\mathsf{SEQUENCE}\,\Phi')\end{array}}{\langle \mathcal{A} \rangle \Gamma \models (\psi, \mathsf{Const}\,\tilde{V}) : \tau\,\mathrm{T}}\ [16]$$

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}, \sigma', s') \\ s' \lhd \mathsf{Some}\,\mathsf{OPTIONAL} \qquad \langle \mathcal{A} \rangle \Gamma \models \tilde{v} : \tau\ (\mathsf{SEQUENCE}\,\Phi')\end{array}}{\langle \mathcal{A} \rangle \Gamma \models \tilde{v} : \tau\ (\mathsf{SEQUENCE}\,([\varphi'] \sqcup \Phi'))}\ [17]$$

$$\frac{\mathrm{T} \lhd \mathsf{SEQUENCE}\,[\,] \qquad \psi \lhd (\mathsf{UNIVERSAL}, \mathsf{Imm}\,16)}{\langle \mathcal{A} \rangle \Gamma \models (\psi, \mathsf{Const}\,[\,]) : [\mathsf{Tag}\ \psi\ \mathsf{IMPLICIT}]\ \mathrm{T}}\ [18]$$

Nous allons prouver qu'aucune d'elles ne peut en réalité être appliquée. Tout d'abord, **HF.1** rend impossible l'application de la règle [18] (qui ne filtre que $\Phi \lhd [\,]$). D'autre part, **H.0.1** implique

— **H.2** : $\neg(\langle\mathcal{A}\rangle\Gamma \vdash e' \parallel (\tau', \mathrm{T}'))$

La propriété de symétrie de $\parallel$ (propriété 8.2.2.3 page 162) et **H.2** impliquent :

— **H.3** : $\neg(\langle\mathcal{A}\rangle\Gamma \vdash (\tau', \mathrm{T}') \parallel e')$

Enfin, **H.1** et **H.3** et le théorème 8.2.2.4 page 163 impliquent

— **H.4** : $\neg(\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}(\tilde{v}') \sqsubseteq (\tau', \mathrm{T}'))$

Or **H.4** est la négation d'une prémisse de la règle [16] page 166, qui ne peut donc s'appliquer. Enfin, **HF.0** rend impossible l'application de la règle [17] page 166 dont l'une des prémisses est :

— $s' \lhd \mathsf{Some\ OPTIONAL}$

En conclusion, puisque les seules règles pouvant dériver $\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{SEQUENCE}\,\Phi)$ ne peuvent s'appliquer, cela signifie que **C** est vraie. △

— $\Phi \lhd \varphi' :: \Phi'$
et $\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s')$
et $\tilde{v} \lhd (\psi, \mathsf{Const}\,(\tilde{v}' :: \tilde{V}'))$
et $s' \lhd \mathsf{Some\ OPTIONAL}$

Distinguons des hypothèses de filtre :

— **HF.0** : $s' \lhd \mathsf{Some\ OPTIONAL}$

— **HF.1** : $\Phi \lhd \varphi' :: \Phi'$

Les autres hypothèses sont

— **H.0** : $\langle\mathcal{A}\rangle\Gamma, \{e'\} \cup \mathrm{E} \vdash_{\mathcal{T}} \mathsf{SEQUENCE}\,\Phi$. Les propositions **HF.0** et **HF.1** font que l'on applique la règle [6] page 198 :

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \qquad s' \not\lhd \mathsf{None} \\ \langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{T}} \mathrm{T}' \qquad \forall e \in \mathrm{E}.\neg(\langle\mathcal{A}\rangle\Gamma \vdash e \parallel (\tau', \mathrm{T}')) \\ \langle\mathcal{A}\rangle\Gamma, \{(\tau', \mathrm{T}')\} \cup \mathrm{E} \vdash_{\mathcal{T}} \mathsf{SEQUENCE}\,\Phi'\end{array}}{\langle\mathcal{A}\rangle\Gamma, \mathrm{E} \vdash_{\mathcal{T}} \mathsf{SEQUENCE}\,(\varphi' :: \Phi')}\ [6]$$

Appliquons-là. Ce faisant nous effectuons la substitution :

— $\mathrm{E} \xleftarrow{\beta} \{e'\} \cup \mathrm{E}$

Étiquetons ensuite ses prémisses non redondantes avec les hypothèses de filtre (ou moins précises) :

— **H.0.0** : $\langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{T}} \mathrm{T}'$

— **H.0.1** : $\forall e \in \{e'\} \cup \mathrm{E}.\neg(\langle\mathcal{A}\rangle\Gamma \vdash e \parallel (\tau', \mathrm{T}'))$

— **H.0.2** : $\langle\mathcal{A}\rangle\Gamma, \{(\tau', \mathrm{T}')\} \cup (\{e'\} \cup \mathrm{E}) \vdash_{\mathcal{T}} \mathsf{SEQUENCE}\,\Phi'$

— **H.1** : $\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}') \sqsubseteq e'$

Prouvons

— **C** : $\neg(\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{SEQUENCE}\,\Phi))$

Les seules règles dérivant des jugements sur les types SEQUENCE sont celles numérotées [16] page 165, [17] page 165 et [18] page 165 :

$$\frac{\begin{array}{c}\mathrm{T} \lhd \mathsf{SEQUENCE}\,(\varphi' :: \Phi') \qquad \varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \\ \tilde{V} \lhd \tilde{v}' :: \tilde{V}' \qquad \langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}(\tilde{v}') \sqsubseteq (\tau', \mathrm{T}') \qquad \langle\mathcal{A}\rangle\Gamma \models \tilde{v}' : \tau'\,\mathrm{T}' \\ \langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Const}\,\tilde{V}') : \tau\ (\mathsf{SEQUENCE}\,\Phi')\end{array}}{\langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Const}\,\tilde{V}) : \tau\,\mathrm{T}}\ [16]$$

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}, \sigma', s') \\ s' \lhd \mathsf{Some}\,\mathsf{OPTIONAL} \qquad \langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{SEQUENCE}\,\Phi')\end{array}}{\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{SEQUENCE}\,([\varphi'] \sqcup \Phi'))}\ [17]$$

$$\frac{\mathrm{T} \lhd \mathsf{SEQUENCE}\,[\,] \qquad \psi \lhd (\mathsf{UNIVERSAL}, \mathsf{Imm}\,16)}{\langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Const}\,[\,]) : [\mathsf{Tag}\ \psi\ \mathsf{IMPLICIT}]\ \mathrm{T}}\ [18]$$

Nous allons prouver qu'aucune d'elles ne peut en réalité être appliquée. Tout d'abord, **HF.1** rend impossible l'application de la règle [18] (qui ne filtre que $\Phi \lhd [\,]$). D'autre part, **H.0.1** implique
— **H.2** : $\neg(\langle\mathcal{A}\rangle\Gamma \vdash e' \parallel (\tau', \mathrm{T}'))$
La propriété de symétrie de $\parallel$ (propriété 8.2.2.3 page 162) et **H.2** impliquent :
— **H.3** : $\neg(\langle\mathcal{A}\rangle\Gamma \vdash (\tau', \mathrm{T}') \parallel e')$
Enfin, les propositions **H.1** et **H.3**, ainsi que le théorème 8.2.2.4 page 163 impliquent
— **H.4** : $\neg(\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}(\tilde{v}') \sqsubseteq (\tau', \mathrm{T}'))$
Or **H.4** est la négation d'une prémisse de la règle [16], qui ne peut donc s'appliquer. Il ne nous reste qu'à prouver que la règle [17] est inapplicable. Pour ce faire, reprenons l'énoncé du théorème en y effectuant les renommages :
— $\mathrm{E} \xleftarrow{\alpha} \{(\tau', \mathrm{T}')\} \cup \mathrm{E}$
— $\Phi \xleftarrow{\alpha} \Phi'$
Nous obtenons alors :

Soit	$\tilde{v} \lhd (\psi, \mathsf{Const}\,(\tilde{v}' :: \tilde{V}'))$
Si	$\langle\mathcal{A}\rangle\Gamma, \{e'\} \cup (\{(\tau', \mathrm{T}')\} \cup \mathrm{E}) \vdash_{\mathcal{T}} \mathsf{SEQUENCE}\,\Phi'$
et	$\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}') \sqsubseteq e'$
alors	$\neg(\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{SEQUENCE}\,\Phi'))$

Nous reconnaissons en cet énoncé une instance de l'hypothèse d'induction, car ses prémisses sont, dans l'ordre d'écriture, **H.0.2** et **H.1**. En l'appliquant il vient donc :
— **H.5** : $\neg(\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{SEQUENCE}\,\Phi'))$
Or **H.5** est la négation d'une prémisse de la règle [17], qui ne peut donc être appliquée. En conclusion, puisque les seules règles pouvant dériver $\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{SEQUENCE}\,\Phi)$ ne peuvent s'appliquer, cela signifie que **C** est vraie. △

— $\Phi \lhd \varphi' :: \Phi'$
et $\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s')$
et $\tilde{v} \lhd (\psi, \mathsf{Const}\,(\tilde{v}' :: \tilde{V}'))$
et $s' \lhd \mathsf{Some}\,(\mathsf{DEFAULT}\ \_)$

Distinguons deux des hypothèses de filtre :
— **HF.0** : $s' \lhd \mathsf{Some}\,(\mathsf{DEFAULT}\ \_)$
— **HF.1** : $\Phi \lhd \varphi' :: \Phi'$
Les autres hypothèses sont
— **H.0** : $\langle\mathcal{A}\rangle\Gamma, \{e'\} \cup \mathrm{E} \vdash_{\mathcal{T}} \mathsf{SEQUENCE}\,\Phi$. Les propositions **HF.0** et **HF.1** font que l'on applique la règle [6] page 198 :

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \qquad s' \not\lhd \mathsf{None} \\ \langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{T}} \mathrm{T}' \qquad \forall e \in \mathrm{E}.\neg(\langle\mathcal{A}\rangle\Gamma \vdash e \parallel (\tau', \mathrm{T}')) \\ \langle\mathcal{A}\rangle\Gamma, \{(\tau', \mathrm{T}')\} \cup \mathrm{E} \vdash_{\mathcal{T}} \mathsf{SEQUENCE}\,\Phi'\end{array}}{\langle\mathcal{A}\rangle\Gamma, \mathrm{E} \vdash_{\mathcal{T}} \mathsf{SEQUENCE}\,(\varphi' :: \Phi')}\ [6]$$

Appliquons-là. Ce faisant nous effectuons la substitution :
— $\mathrm{E} \xleftarrow{\beta} \{e'\} \cup \mathrm{E}$
Étiquetons ensuite ses prémisses non redondantes avec les hypothèses de filtre (ou moins précises) :
— **H.0.0** : $\langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{T}} \mathrm{T}'$
— **H.0.1** : $\forall e \in \{e'\} \cup \mathrm{E}.\neg(\langle\mathcal{A}\rangle\Gamma \vdash e \parallel (\tau', \mathrm{T}'))$
— **H.0.2** : $\langle\mathcal{A}\rangle\Gamma, \{(\tau', \mathrm{T}')\} \cup (\{e'\} \cup \mathrm{E}) \vdash_{\mathcal{T}} \mathsf{SEQUENCE}\,\Phi'$
— **H.1** : $\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}') \sqsubseteq e'$
Prouvons
— **C** : $\neg(\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{SEQUENCE}\,\Phi))$
Les seules règles dérivant des jugements sur les types SEQUENCE sont celles numérotées [16] page 165, [17] page 165 et [18] page 165 :

$$\frac{\begin{array}{c}\mathrm{T} \lhd \mathsf{SEQUENCE}\,(\varphi' :: \Phi') \qquad \varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \\ \tilde{V} \lhd \tilde{v}' :: \tilde{V}' \qquad \langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}(\tilde{v}') \sqsubseteq (\tau', \mathrm{T}') \qquad \langle\mathcal{A}\rangle\Gamma \models \tilde{v}' : \tau'\ \mathrm{T}' \\ \langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Const}\,\tilde{V}') : \tau\ (\mathsf{SEQUENCE}\,\Phi')\end{array}}{\langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Const}\,\tilde{V}) : \tau\ \mathrm{T}}\ [16]$$

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}, \sigma', s') \\ s' \lhd \mathsf{Some}\,\mathsf{OPTIONAL} \qquad \langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{SEQUENCE}\,\Phi')\end{array}}{\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{SEQUENCE}\,([\varphi'] \sqcup \Phi'))}\ [17]$$

$$\frac{\mathrm{T} \lhd \mathsf{SEQUENCE}\,[\,] \qquad \psi \lhd (\mathsf{UNIVERSAL}, \mathsf{Imm}\,16)}{\langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Const}\,[\,]) : [\mathsf{Tag}\ \psi\ \mathsf{IMPLICIT}]\ \mathrm{T}}\ [18]$$

Nous allons prouver qu'aucune d'elles ne peut en réalité être appliquée. Tout d'abord **HF.1** rend impossible l'application de la règle [18] (qui ne filtre que $\Phi \lhd [\,]$). D'autre part, **H.0.1** implique
— **H.2** : $\neg(\langle\mathcal{A}\rangle\Gamma \vdash e' \parallel (\tau', \mathrm{T}'))$
La propriété de symétrie de $\parallel$ (propriété 8.2.2.3 page 162) et **H.2** impliquent :
— **H.3** : $\neg(\langle\mathcal{A}\rangle\Gamma \vdash (\tau', \mathrm{T}') \parallel e')$
Enfin, **H.1** et **H.3** et le théorème 8.2.2.4 page 163 impliquent
— **H.4** : $\neg(\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}(\tilde{v}') \sqsubseteq (\tau', \mathrm{T}'))$
Or **H.4** est la négation d'une prémisse de la règle [16] page 166, qui ne peut donc s'appliquer. Enfin, **HF.0** rend impossible l'application de la règle [17] page 166 dont l'une des prémisses est :
— $s' \lhd \mathsf{Some}\ \mathsf{OPTIONAL}$
En conclusion, puisque les seules règles pouvant dériver $\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{SEQUENCE}\,\Phi)$ ne peuvent s'appliquer, cela signifie que **C** est vraie. △

— $\Phi \lhd [\,]$
et $\tilde{v} \lhd (\psi, \mathsf{Const}\,(\tilde{v}' :: \tilde{V}'))$

Les autres hypothèse sont
— **H.0** : $\langle\mathcal{A}\rangle\Gamma, \{e'\} \cup \mathrm{E} \vdash_{\mathcal{T}} \mathsf{SEQUENCE}\,\Phi$
— **H.1** : $\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}') \sqsubseteq e'$
Prouvons
— **C** : $\neg(\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{SEQUENCE}\,\Phi))$
Aucune règle ne permet de dériver $\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{SEQUENCE}\,\Phi)$.
Cela signifie que **C** est vraie. △

□

Lemme 8.7.4.

Si	$\langle\mathcal{A}\rangle\Gamma, E \vdash_{\mathcal{T}} \mathsf{SET}\,\Phi$
et	$\langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Const}\,\tilde{V}) : \tau\ (\mathsf{SET}\,\Phi)$
alors	$\forall e \in E. \forall \tilde{v}' \in \tilde{V}. \neg(\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}') \sqsubseteq e)$

Preuve du lemme 8.7.4

La démonstration se fera par induction.
— $\Phi \lhd [\varphi'] \sqcup \Phi'$ et $\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s')$
et $\tilde{v} \lhd (\psi, \mathsf{Const}\,\tilde{V})$ et $\tilde{V} \lhd [\tilde{v}'] \sqcup \tilde{V}'$

Distinguons une des hypothèses de filtre :

— **HF.0** : $\tilde{V} \lhd [\tilde{v}'] \sqcup \tilde{V}'$

Les autres hypothèses sont

— **H.0** : $\langle\mathcal{A}\rangle\Gamma, \mathrm{E} \vdash_{\mathcal{T}} \mathsf{SET}\,\Phi$. C'est la règle [4] page 197 qui s'applique :

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \\ \langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{T}} \mathrm{T}' \qquad \forall e \in \mathrm{E}.\neg(\langle\mathcal{A}\rangle\Gamma \vdash e \parallel (\tau', \mathrm{T}')) \\ \langle\mathcal{A}\rangle\Gamma, \{(\tau', \mathrm{T}')\} \cup \mathrm{E} \vdash_{\mathcal{T}} \mathsf{SET}\,\Phi'\end{array}}{\langle\mathcal{A}\rangle\Gamma, \mathrm{E} \vdash_{\mathcal{T}} \mathsf{SET}\,([\varphi'] \sqcup \Phi')}\ [4]$$

Appliquons-là directement (i.e. toutes les variables de la règle sont liées aux variables de même nom dans l'environnement du cas courant de la preuve). Étiquetons ses prémisses :

— **H.0.0** : $\langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{T}} \mathrm{T}'$

— **H.0.1** : $\forall e \in \mathrm{E}.\neg(\langle\mathcal{A}\rangle\Gamma \vdash e \parallel (\tau', \mathrm{T}'))$

— **H.0.2** : $\langle\mathcal{A}\rangle\Gamma, \{(\tau', \mathrm{T}')\} \cup \mathrm{E} \vdash_{\mathcal{T}} \mathsf{SET}\,\Phi'$

— **H.1** : $\langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Const}\,\tilde{V}) : \tau\ \mathsf{SET}\,\Phi$. La règle [19] page 166 peut s'appliquer :

$$\frac{\begin{array}{cc}\mathrm{T} \lhd \mathsf{SET}\,([\varphi'] \sqcup \Phi') & \varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \\ \tilde{V} \lhd [\tilde{v}'] \sqcup \tilde{V}' & \langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}(\tilde{v}') \sqsubseteq (\tau', \mathrm{T}') \\ \langle\mathcal{A}\rangle\Gamma \models \tilde{v}' : \tau'\,\mathrm{T}' & \langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Const}\,\tilde{V}') : \tau\ (\mathsf{SET}\,\Phi')\end{array}}{\langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Const}\,\tilde{V}) : \tau\,\mathrm{T}}\ [19]$$

Appliquons-là directement et étiquetons ses prémisses non redondantes avec les hypothèses de filtre :

— **H.1.0** : $\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}(\tilde{v}') \sqsubseteq (\tau', \mathrm{T}')$

— **H.1.1** : $\langle\mathcal{A}\rangle\Gamma \models \tilde{v}' : \tau'\,\mathrm{T}'$

— **H.1.2** : $\langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Const}\,\tilde{V}') : \tau\ (\mathsf{SET}\,\Phi')$

Prouvons

— **C** : $\forall e \in \mathrm{E}.\forall \tilde{v}' \in \tilde{V}.\neg(\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}') \sqsubseteq e)$

Reprenons l'énoncé du théorème en y effectuant les renommages suivants :

— $\mathrm{E} \xleftarrow{\alpha} \{(\tau', \mathrm{T}')\} \cup \mathrm{E}$

— $\tilde{v}' \xleftarrow{\alpha} \tilde{v}''$

— $\tilde{V} \xleftarrow{\alpha} \tilde{V}'$

Il devient alors :

Si $\quad \langle\mathcal{A}\rangle\Gamma, \{(\tau', \mathrm{T}')\} \cup \mathrm{E} \vdash_{\mathcal{T}} \mathsf{SET}\,\Phi$

et $\quad \langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Const}\,\tilde{V}') : \tau\ (\mathsf{SET}\,\Phi)$

alors $\quad \forall e \in \{(\tau', \mathrm{T}')\} \cup \mathrm{E}.\forall \tilde{v}'' \in \tilde{V}'.\neg(\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}'') \sqsubseteq e)$

Nous reconnaissons en cet énoncé une instance de l'hypothèse d'induction, car ses prémisses sont, dans l'ordre d'écriture, **H.0.2** et **H.1.2**. En l'appliquant il vient donc :

— **H.2** : $\forall e \in \{(\tau', \mathrm{T}')\} \cup \mathrm{E}.\forall \tilde{v}'' \in \tilde{V}'.\neg(\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}'') \sqsubseteq e)$

La proposition **H.2** implique

— **H.3** : $\forall e \in \mathrm{E}.\forall \tilde{v}'' \in \tilde{V}'.\neg(\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}(\tilde{v}'') \sqsubseteq e)$

D'autre part, **H.1.0**, **H.0.1** et le théorème 8.2.2.4 page 163 impliquent

— **H.4** : $\forall e \in \mathrm{E}.\neg(\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}(\tilde{v}') \sqsubseteq e)$

La proposition (**H.3** $\wedge$ **H.4**) est équivalente à

— **H.5** : $\forall e \in \mathrm{E}.(\neg(\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}(\tilde{v}') \sqsubseteq e) \wedge \forall \tilde{v}'' \in \tilde{V}'.\neg(\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}(\tilde{v}'') \sqsubseteq e))$

Et **H.5** est équivalente à

— **H.6** : $\forall e \in \mathrm{E}.\forall \tilde{v}'' \in [\tilde{v}'] \sqcup \tilde{V}'.\neg(\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}(\tilde{v}'') \sqsubseteq e)$

Et **H.6** et **HF.0** impliquent **C**. △

— $\Phi \lhd [\varphi'] \sqcup \Phi'$
et $\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s')$
et $s' \lhd \mathsf{Some\ OPTIONAL}$

Les autres hypothèses sont

— **H.0** : $\langle\mathcal{A}\rangle\Gamma, \mathrm{E} \vdash_{\mathcal{T}} \mathsf{SET}\,\Phi$. C'est la règle [4] page 197 qui s'applique :

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \\ \langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{T}} \mathrm{T}' \qquad \forall e \in \mathrm{E}.\neg(\langle\mathcal{A}\rangle\Gamma \vdash e \parallel (\tau', \mathrm{T}')) \\ \langle\mathcal{A}\rangle\Gamma, \{(\tau', \mathrm{T}')\} \cup \mathrm{E} \vdash_{\mathcal{T}} \mathsf{SET}\,\Phi'\end{array}}{\langle\mathcal{A}\rangle\Gamma, \mathrm{E} \vdash_{\mathcal{T}} \mathsf{SET}\,([\varphi'] \sqcup \Phi')}\ [4]$$

Appliquons-là directement (i.e. toutes les variables de la règle sont liées aux variables de même nom dans l'environnement du cas courant de la preuve). Étiquetons ses prémisses :

— **H.0.0** : $\langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{T}} \mathrm{T}'$

— **H.0.1** : $\forall e \in \mathrm{E}.\neg(\langle\mathcal{A}\rangle\Gamma \vdash e \parallel (\tau', \mathrm{T}'))$

— **H.0.2** : $\langle\mathcal{A}\rangle\Gamma, \{(\tau', \mathrm{T}')\} \cup \mathrm{E} \vdash_{\mathcal{T}} \mathsf{SET}\,\Phi'$

— **H.1** : $\langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Const}\,\tilde{V}) : \tau\ (\mathsf{SET}\,\Phi)$. La règle [20] page 166 peut être appliquée :

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}, \sigma', s') \\ s' \lhd \mathsf{Some\,OPTIONAL} \qquad \langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{SET}\,\Phi')\end{array}}{\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{SET}\,([\varphi'] \sqcup \Phi'))}\ [20]$$

Appliquons directement la règle (i.e. toutes les variables de la règle sont liées aux variables de même nom dans l'environnement du cas courant de la preuve). Étiquetons ses prémisses non redondantes :

— **H.1.0** : $\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{SET}\,\Phi')$

Prouvons
— $\mathbf{C} : \forall e \in \mathrm{E}.\forall \tilde{v}' \in \tilde{V}.\neg(\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}') \sqsubseteq e)$
Reprenons l'énoncé du lemme et effectuons-y les renommages suivants :
— $\mathrm{E} \xleftarrow{\alpha} \{(\tau', \mathrm{T}')\} \cup \mathrm{E}$
— $\Phi \xleftarrow{\alpha} \Phi'$
Nous obtenons alors :

$$\begin{array}{ll} \mathit{Si} & \langle\mathcal{A}\rangle\Gamma, \{(\tau', \mathrm{T}')\} \cup \mathrm{E} \vdash_{\mathcal{T}} \mathsf{SET}\,\Phi' \\ \mathit{et} & \langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Const}\,\tilde{V}) : \tau\ (\mathsf{SET}\,\Phi') \\ \mathit{alors} & \forall e \in \{(\tau', \mathrm{T}')\} \cup \mathrm{E}.\forall \tilde{v}' \in \tilde{V}.\neg(\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}') \sqsubseteq e) \end{array}$$

Nous reconnaissons en cet énoncé une instance de l'hypothèse d'induction, car ses prémisses sont, dans l'ordre d'écriture, **H.0.2** et **H.1.0**. En l'appliquant il vient donc :
— $\mathbf{H.2} : \forall e \in \{(\tau', \mathrm{T}')\} \cup \mathrm{E}.\forall \tilde{v}' \in \tilde{V}.\neg(\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}') \sqsubseteq e)$
Et **H.2** implique immédiatement **C**. △

— $\Phi \lhd [\,]$
et $\tilde{v} \lhd (\psi, \mathsf{Const}\,\tilde{V})$
et $\tilde{V} \lhd [\,]$

Distinguons une des hypothèses de filtre :
— $\mathbf{HF.0} : \tilde{V} \lhd [\,]$
Les autres hypothèses sont
— $\mathbf{H.0} : \langle\mathcal{A}\rangle\Gamma, \mathrm{E} \vdash_{\mathcal{T}} \mathsf{SET}\,\Phi$
— $\mathbf{H.1} : \langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Const}\,\tilde{V}) : \tau\ (\mathsf{SET}\,\Phi)$
Prouvons
— $\mathbf{C} : \forall e \in \mathrm{E}.\forall \tilde{v}' \in \tilde{V}.\neg(\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}') \sqsubseteq e)$
HF.0 implique trivialement **C**. △

□

Lemme 8.7.5.

$$\begin{array}{ll} \mathit{Soit} & \tilde{v} \lhd (\psi, \mathsf{Const}\,([\tilde{v}'] \sqcup \tilde{V}')) \\ \mathit{Si} & \langle\mathcal{A}\rangle\Gamma, \{e'\} \cup E \vdash_{\mathcal{T}} \mathsf{SET}\,\Phi \\ \mathit{et} & \langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}') \sqsubseteq e' \\ \mathit{alors} & \neg(\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{SET}\,\Phi)) \end{array}$$

Preuve du lemme 8.7.5

Soient

— **H.0** : $\tilde{v} \lhd (\psi, \mathsf{Const}\, \tilde{V})$
— **H.1** : $\tilde{V} \lhd [\tilde{v}'] \sqcup \tilde{V}'$

Reprenons l'énoncé du lemme 8.7.4 page 205 et effectuons-y le renommage suivant :

— $\mathrm{E} \xleftarrow{\alpha} \{e'\} \cup \mathrm{E}$

Il devient alors :

$$\begin{array}{ll} Si & \langle \mathcal{A} \rangle \Gamma, \{e'\} \cup \mathrm{E} \vdash_{\mathcal{T}} \mathsf{SET}\, \Phi \\ et & \langle \mathcal{A} \rangle \Gamma \models (\psi, \mathsf{Const}\, \tilde{V}) : \tau\ (\mathsf{SET}\, \Phi) \\ alors & \forall e \in \{e'\} \cup \mathrm{E}. \forall \tilde{v}' \in \tilde{V}. \neg(\langle \mathcal{A} \rangle \Gamma \vdash \mathcal{D}\, (\tilde{v}')\ \sqsubseteq\ e) \end{array}$$

Évaluons partiellement ce lemme en choisissant $e = e'$:

$$\begin{array}{ll} Si & \langle \mathcal{A} \rangle \Gamma, \{e'\} \cup \mathrm{E} \vdash_{\mathcal{T}} \mathsf{SET}\, \Phi \\ et & \langle \mathcal{A} \rangle \Gamma \models (\psi, \mathsf{Const}\, \tilde{V}) : \tau\ (\mathsf{SET}\, \Phi) \\ alors & \forall \tilde{v}' \in \tilde{V}. \neg(\langle \mathcal{A} \rangle \Gamma \vdash \mathcal{D}\, (\tilde{v}')\ \sqsubseteq\ e') \end{array}$$

Cet énoncé est équivalent à :

$$\begin{array}{ll} Si & \langle \mathcal{A} \rangle \Gamma, \{e'\} \cup \mathrm{E} \vdash_{\mathcal{T}} \mathsf{SET}\, \Phi \\ et & \langle \mathcal{A} \rangle \Gamma \models (\psi, \mathsf{Const}\, \tilde{V}) : \tau\ (\mathsf{SET}\, \Phi) \\ alors & \neg(\tilde{V} \lhd [\tilde{v}'] \sqcup \tilde{V}' \Rightarrow \langle \mathcal{A} \rangle \Gamma \vdash \mathcal{D}\, (\tilde{v}')\ \sqsubseteq\ e') \end{array}$$

En tenant maintenant compte de **H.0** et **H.1**, ce dernier énoncé se réécrit :

$$\begin{array}{ll} Si & \langle \mathcal{A} \rangle \Gamma, \{e'\} \cup \mathrm{E} \vdash_{\mathcal{T}} \mathsf{SET}\, \Phi \\ et & \langle \mathcal{A} \rangle \Gamma \models (\psi, \mathsf{Const}\, \tilde{V}) : \tau\ (\mathsf{SET}\, \Phi) \\ alors & \neg(\tilde{v} \lhd (\psi, \mathsf{Const}\, ([\tilde{v}'] \sqcup \tilde{V}')) \Rightarrow \langle \mathcal{A} \rangle \Gamma \vdash \mathcal{D}\, (\tilde{v}')\ \sqsubseteq\ e') \end{array}$$

Prenons-en finalement une contraposition et nous obtenons la preuve de notre lemme 8.7.5 page ci-contre. □

Preuve du théorème 8.7.1

Rappelons l'énoncé du théorème :

> *S'il existe une preuve qu'une valeur codée est d'un type bien étiqueté, alors cette preuve est unique.*

Les hypothèses sont donc :

— **H.0** : $\langle \mathcal{A} \rangle \Gamma \vdash_{\mathcal{T}} \mathrm{T}$
— **H.1** : $\langle \mathcal{A} \rangle \Gamma \models \tilde{v} : \tau\, \mathrm{T}$

La preuve se fait par induction sur la forme de T. Nous allons prouver qu'à chaque étape de la dérivation, ou bien aucune règle n'est applicable, ou bien une seule instance d'une seule règle est applicable, grâce à **H.0** et **H.1**. Tout d'abord on remarque que la seule façon de prouver **H.0** est de prouver

— **H.0bis** : $\langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{L}} \mathrm{T}$.

par la règle [1] page 197 :

$$\frac{\langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{T}} \mathrm{T}}{\langle\mathcal{A}\rangle\Gamma \vdash_{\mathcal{T}} \mathrm{T}}\ [1]$$

Nous remplaçons alors **H.0** par **H.0bis** dans l'énoncé du théorème. Les seules règles à examiner de près sont [14] page 165 et [15] page 165 (pour le type CHOICE), [16] page 165 à [18] page 165 (pour le type SEQUENCE) et [19] page 166 à [21] page 166 (pour le type SET).

— $\mathrm{T} \lhd \mathsf{CHOICE}\,([f'] \sqcup \mathcal{F}')$
et $f' \lhd (l', \tau', \mathrm{T}', \sigma', s')$

Les autres hypothèses sont

— **H.0bis** : $\langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{T}} \mathrm{T}$. La règle [3] page 197 peut être appliquée :

$$\frac{\begin{array}{c} f' \lhd (l', \tau', \mathrm{T}', \sigma', s') \\ \langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{T}} \mathrm{T}' \qquad \forall e \in \mathrm{E}.\neg(\langle\mathcal{A}\rangle\Gamma \vdash e \parallel (\tau', \mathrm{T}')) \\ \langle\mathcal{A}\rangle\Gamma, \{(\tau', \mathrm{T}')\} \cup \mathrm{E} \vdash_{\mathcal{T}} \mathsf{CHOICE}\,\mathcal{F}' \end{array}}{\langle\mathcal{A}\rangle\Gamma, \mathrm{E} \vdash_{\mathcal{T}} \mathsf{CHOICE}\,([f'] \sqcup \mathcal{F}')}\ [3]$$

Appliquons-là directement (i.e. toutes les variables de la règle sont liées aux variables de même nom dans l'environnement du cas courant de la preuve) et étiquetons ses prémisses non redondantes avec les hypothèses de filtre :

— **H.0bis.0** : $\langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{T}} \mathrm{T}'$
— **H.0bis.1** : $\forall e \in \mathrm{E}.\neg(\langle\mathcal{A}\rangle\Gamma \vdash e \parallel (\tau', \mathrm{T}'))$
— **H.0bis.2** : $\langle\mathcal{A}\rangle\Gamma, \{(\tau', \mathrm{T}')\} \cup \mathrm{E} \vdash_{\mathcal{T}} \mathsf{CHOICE}\,\mathcal{F}'$

— **H.1** : $\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ \mathrm{T}$
Les règles [14] page 165 ou [15] page 165 peuvent être appliquées :

$$\frac{\begin{array}{c} f' \lhd (l', \tau', \mathrm{T}', \sigma', s') \\ \langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}(\tilde{v}) \sqsubseteq (\tau', \mathrm{T}') \qquad \langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau'\ \mathrm{T}' \end{array}}{\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : [\,]\ (\mathsf{CHOICE}\,([f'] \sqcup \mathcal{F}'))}\ [14]$$

$$\frac{\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{CHOICE}\,\mathcal{F}')}{\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{CHOICE}\,([f'] \sqcup \mathcal{F}'))}\ [15]$$

Nous constatons d'abord que si $\neg(\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}(\tilde{v}) \sqsubseteq (\tau', \mathrm{T}'))$, alors la seule règle applicable est [15]. Supposons donc maintenant :

— **H.2** : $\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}(\tilde{v}) \sqsubseteq (\tau', \mathrm{T}')$

Prouvons que la seule règle pouvant éventuellement être appliquée est [14]. Reprenons l'énoncé du théorème 8.7.2 page 198 en y effectuant les renommages suivants :

— $e' \xleftarrow{\alpha} (\tau', \mathrm{T}')$

— $\mathcal{F} \xleftarrow{\alpha} \mathcal{F}'$

Nous obtenons alors :

$$\begin{array}{ll} Si & \langle\mathcal{A}\rangle\Gamma, \{(\tau', \mathrm{T}')\} \cup \mathrm{E} \vDash_{\mathcal{T}} \mathsf{CHOICE}\,\mathcal{F}' \\ et & \langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}) \sqsubseteq (\tau', \mathrm{T}') \\ alors & \neg(\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{CHOICE}\,\mathcal{F}')) \end{array}$$

Appliquons ce théorème à **H.0bis.2** et **H.2**. Il vient :

— **H.2** : $\neg(\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{CHOICE}\,\mathcal{F}'))$

Or **H.2** est la négation de la prémisse de la règle [15], qui ne peut donc être appliquée. △

— $\mathrm{T} \lhd \mathsf{SET}\,([\varphi'] \sqcup \Phi')$ et $\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s')$

Les autres hypothèses sont

— **H.0bis** : $\langle\mathcal{A}\rangle\Gamma, \{\} \vDash_{\mathcal{T}} \mathrm{T}$. La règle [4] page 197 peut être appliquée :

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \\ \langle\mathcal{A}\rangle\Gamma, \{\} \vDash_{\mathcal{T}} \mathrm{T}' \qquad \forall e \in \mathrm{E}.\neg(\langle\mathcal{A}\rangle\Gamma \vdash e \parallel (\tau', \mathrm{T}')) \\ \langle\mathcal{A}\rangle\Gamma, \{(\tau', \mathrm{T}')\} \cup \mathrm{E} \vDash_{\mathcal{T}} \mathsf{SET}\,\Phi'\end{array}}{\langle\mathcal{A}\rangle\Gamma, \mathrm{E} \vDash_{\mathcal{T}} \mathsf{SET}\,([\varphi'] \sqcup \Phi')}\ [4]$$

Appliquons-là directement et étiquetons ses prémisses non redondantes :

— **H.0bis.0** : $\langle\mathcal{A}\rangle\Gamma, [\,] \vDash_{\mathcal{T}} \mathrm{T}'$

— **H.0bis.1** : $\forall e \in \mathrm{E}.\neg(\langle\mathcal{A}\rangle\Gamma \vdash e \parallel (\tau', \mathrm{T}'))$

— **H.0bis.2** : $\langle\mathcal{A}\rangle\Gamma, \{(\tau', \mathrm{T}')\} \cup \mathrm{E} \vDash_{\mathcal{T}} \mathsf{SET}\,\Phi'$

— **H.1** : $\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\,\mathrm{T}$. Les seules règles à examiner sont [19] page 166 et [20] page 166 :

$$\frac{\begin{array}{cc}\mathrm{T} \lhd \mathsf{SET}\,([\varphi'] \sqcup \Phi') & \varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \\ \tilde{V} \lhd [\tilde{v}'] \sqcup \tilde{V}' & \langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}(\tilde{v}') \sqsubseteq (\tau', \mathrm{T}') \\ \langle\mathcal{A}\rangle\Gamma \models \tilde{v}' : \tau'\,\mathrm{T}' & \langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Const}\,\tilde{V}') : \tau\ (\mathsf{SET}\,\Phi')\end{array}}{\langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Const}\,\tilde{V}) : \tau\,\mathrm{T}}\ [19]$$

$$\frac{\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}, \sigma', s') \qquad s' \lhd \mathsf{Some}\,\mathsf{OPTIONAL} \qquad \langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{SET}\,\Phi')}{\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{SET}\,([\varphi'] \sqcup \Phi'))}\ [20]$$

car la règle [21] page 166 ne filtre que $\Phi \lhd [\,]$.
Premièrement, si $\tilde{v} \not\lhd (\psi, \mathsf{Const}\,\tilde{V})$, alors la seule règle pouvant s'appliquer serait [20], ce qui démontrerait le théorème. Aussi posons :
— **HF.0** : $\tilde{v} \lhd (\psi, \mathsf{Const}\,\tilde{V})$
De même si $\tilde{V} \not\lhd [\tilde{v}'] \sqcup \tilde{V}'$, alors la seule règle pouvant éventuellement être appliquée serait [20], ce qui démontrerait aussi le théorème. Supposons donc pour les sous-cas restant :
— **H.4** : $\tilde{V} \lhd [\tilde{v}'] \sqcup \tilde{V}'$
Les propositions **HF.0** et **H.4** impliquent
— **H.5** : $\tilde{v} \lhd (\psi, \mathsf{Const}\,([\tilde{v}'] \sqcup \tilde{V}'))$
Maintenant, les règles [19] et [20] peuvent *a priori* être appliquées, mais *nous allons montrer que seule la règle* [19] *est éventuellement applicable*. Appliquons directement la règle [19] et étiquetons ses prémisses non redondantes avec les hypothèses précédentes :
— **H.1.0** : $\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}(\tilde{v}') \sqsubseteq (\tau', \mathrm{T}')$
— **H.1.1** : $\langle\mathcal{A}\rangle\Gamma \models \tilde{v}' : \tau'\,\mathrm{T}'$
— **H.1.2** : $\langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Const}\,\tilde{V}') : \tau\ (\mathsf{SET}\,\Phi')$
Nous devons considérer deux aspects :

1. Nous devons prouver que **H.0** et **H.1bis** permettent de conclure que
   — **C.0** : $\forall \tilde{v}'' \in \tilde{V}'.\neg(\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}(\tilde{v}'') \sqsubseteq (\tau', \mathrm{T}'))$
   En effet, ainsi **H.4**, qui est une prémisse de **H.1**, serait l'unique projection possible. En d'autres termes :
   — **H.4** $\Rightarrow$ **C.0**
   d'où l'unicité de la projection. Reprenons le lemme 8.7.4 page 205 et effectuons-y les renommages suivants :
   — $\mathrm{E} \xleftarrow{\alpha} \{(\tau', \mathrm{T}')\} \cup \mathrm{E}$
   — $\tilde{v}' \xleftarrow{\alpha} \tilde{v}''$
   — $\tilde{V} \xleftarrow{\alpha} \tilde{V}'$
   — $\Phi \xleftarrow{\alpha} \Phi'$
   Nous obtenons alors le lemme :

Si	$\langle\mathcal{A}\rangle\Gamma, \{(\tau', \mathrm{T}')\} \cup \mathrm{E} \vdash_{\mathcal{T}} \mathsf{SET}\,\Phi'$
et	$\langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Const}\,\tilde{V}') : \tau\ (\mathsf{SET}\,\Phi')$
alors	$\forall e \in \{(\tau', \mathrm{T}')\} \cup \mathrm{E}.\forall \tilde{v}'' \in \tilde{V}'.\neg(\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}'') \sqsubseteq e)$

   Appliquons-le respectivement à **H.0bis.2** et **H.1.2**. Il vient
   — **H.2** : $\forall e \in \{(\tau', \mathrm{T}')\} \cup \mathrm{E}.\forall \tilde{v}'' \in \tilde{V}'.\neg(\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}'') \sqsubseteq e)$

H.2 implique **C.0** (on prend $e = (\tau', \mathrm{T}')$). $\Diamond$

2. *Prouvons que la règle* [20] *page 166 ne peut s'appliquer.* Prouvons la négation de sa prémisse, soit à prouver :
   — **C.1** : $\neg(\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{SET}\,\Phi'))$
   Reprenons l'énoncé du lemme 8.7.5 page 208 et effectuons-y les renommages suivants :
   — $e' \xleftarrow{\alpha} (\tau', \mathrm{T}')$
   — $\Phi \xleftarrow{\alpha} \Phi'$
   Nous obtenons alors :

$$\begin{array}{ll} \mathit{Soit} & \tilde{v} \lhd (\psi, \mathsf{Const}\,([\tilde{v}'] \sqcup \tilde{V}')) \\ \mathit{Si} & \langle\mathcal{A}\rangle\Gamma, \{(\tau', \mathrm{T}')\} \cup \mathrm{E} \vdash_{\mathcal{T}} \mathsf{SET}\,\Phi' \\ \mathit{et} & \langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}') \sqsubseteq e' \\ \mathit{alors} & \neg(\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{SET}\,\Phi')) \end{array}$$

   En appliquant directement ce lemme respectivement à **H.5**, **H.0bis.2** et **H.1.1** il vient **C.1**. $\Diamond$

— $\mathrm{T} \lhd \mathsf{SEQUENCE}\,(\varphi' :: \Phi')$
et $\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s')$

Nous remarquons tout de suite que si $s' \not\lhd \mathsf{Some}\,\mathsf{OPTIONAL}$, alors la seule règle du contrôle sémantique des types applicable serait [19] page 166, ce qui démontrerait ce cas du théorème. Pour étudier les autres cas, supposons donc :
— **HF.0** : $s' \lhd \mathsf{Some}\,\mathsf{OPTIONAL}$
Les autres hypothèses sont
— **H.0bis** : $\langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{T}} \mathrm{T}$. La règle [6] page 198 peut être appliquée :

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \qquad s' \not\lhd \mathsf{None} \\ \langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{T}} \mathrm{T}' \qquad \forall e \in \mathrm{E}.\neg(\langle\mathcal{A}\rangle\Gamma \vdash e \parallel (\tau', \mathrm{T}')) \\ \langle\mathcal{A}\rangle\Gamma, \{(\tau', \mathrm{T}')\} \cup \mathrm{E} \vdash_{\mathcal{T}} \mathsf{SEQUENCE}\,\Phi'\end{array}}{\langle\mathcal{A}\rangle\Gamma, \mathrm{E} \vdash_{\mathcal{T}} \mathsf{SEQUENCE}\,(\varphi' :: \Phi')}\ [6]$$

Appliquons-là et étiquetons ses prémisses non redondantes avec les hypothèses de filtre, ou moins précises quelles (i.e. des instances de celles-ci) :
— **H.0bis.0** : $\langle\mathcal{A}\rangle\Gamma, \{\} \vdash_{\mathcal{T}} \mathrm{T}'$
— **H.0bis.1** : $\forall e \in \mathrm{E}.\neg(\langle\mathcal{A}\rangle\Gamma \vdash e \parallel (\tau', \mathrm{T}'))$
— **H.0bis.2** : $\langle\mathcal{A}\rangle\Gamma, \{(\tau', \mathrm{T}')\} \cup \mathrm{E} \vdash_{\mathcal{T}} \mathsf{SEQUENCE}\,\Phi'$
— **H.1** : $\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ \mathrm{T}$. Les règles à examiner sont [16] page 165 et [17] page 165 :

$$\frac{\begin{array}{c}\mathrm{T} \lhd \mathsf{SEQUENCE}\,(\varphi' :: \Phi') \\ \varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \qquad \tilde{V} \lhd \tilde{v}' :: \tilde{V}' \\ \langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}(\tilde{v}') \sqsubseteq (\tau', \mathrm{T}') \qquad \langle\mathcal{A}\rangle\Gamma \models \tilde{v}' : \tau'\,\mathrm{T}' \\ \langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Const}\,\tilde{V}') : \tau\ (\mathsf{SEQUENCE}\,\Phi')\end{array}}{\langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Const}\,\tilde{V}) : \tau\,\mathrm{T}}\ [16]$$

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}, \sigma', s') \\ s' \lhd \mathsf{Some\,OPTIONAL} \qquad \langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{SEQUENCE}\,\Phi')\end{array}}{\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{SEQUENCE}\,([\varphi'] \sqcup \Phi'))}\ [17]$$

Premièrement, si $\tilde{v} \not\lhd (\psi, \mathsf{Const}\,\tilde{V})$, alors la seule règle applicable serait [17], ce qui prouverait ce cas du théorème. Pour poursuivre l'étude des cas restants, posons donc :
— **HF.1** : $\tilde{v} \lhd (\psi, \mathsf{Const}\,\tilde{V})$
D'autre part, si $\tilde{V} \not\lhd \tilde{v}' :: \tilde{V}'$, alors la seule règle applicable serait [17], ce qui concluerait le cas courant. Pour étudier les cas restants, posons donc :
— **HF.2** : $\tilde{V} \lhd \tilde{v}' :: \tilde{V}'$
Les propositions **HF.1** et **HF.2** impliquent :
— **H.2** : $\tilde{v} \lhd (\psi, \mathsf{Const}\,(\tilde{v}' :: \tilde{V}'))$
Maintenant les règles [16] et[17] peuvent *a priori* être appliquées, mais *nous allons montrer que seule la règle* [16] *est éventuellement applicable.* Appliquons directement la règle [16] et étiquetons ses prémisses non redondantes avec les hypothèses précédentes :
— **H.1.0** : $\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}(\tilde{v}') \sqsubseteq (\tau', \mathrm{T}')$
— **H.1.1** : $\langle\mathcal{A}\rangle\Gamma \models \tilde{v}' : \tau'\,\mathrm{T}'$
— **H.1.2** : $\langle\mathcal{A}\rangle\Gamma \models (\psi, \mathsf{Const}\,\tilde{V}') : \tau\ (\mathsf{SEQUENCE}\,\Phi')$
Prouvons que la règle [17] *page 165 ne peut s'appliquer.* Prouvons la négation d'une de ses prémisses, soit à prouver :
— **C** : $\neg(\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{SEQUENCE}\,\Phi'))$
Considérons l'énoncé du lemme 8.7.3 page 200 et effectuons-y les renommages suivants :
— $e' \xleftarrow{\alpha} (\tau', \mathrm{T}')$
— $\Phi \xleftarrow{\alpha} \Phi'$
Nous obtenons alors :

Soit	$\tilde{v} \lhd (\psi, \mathsf{Const}\,(\tilde{v}' :: \tilde{V}'))$
Si	$\langle\mathcal{A}\rangle\Gamma, \{(\tau', \mathrm{T}')\} \cup \mathrm{E} \vdash_{\mathcal{T}} \mathsf{SEQUENCE}\,\Phi'$
et	$\langle\mathcal{A}\rangle\Gamma \vdash \mathcal{D}\,(\tilde{v}') \sqsubseteq (\tau', \mathrm{T}')$
alors	$\neg(\langle\mathcal{A}\rangle\Gamma \models \tilde{v} : \tau\ (\mathsf{SEQUENCE}\,\Phi'))$

En l'appliquant respectivement à **H.2**, **H.0bis.2** et **H.1.0**, il

vient **C**. ◊△

□

Chapitre 9

Contrôle des sous-types

9.1 Réduction des INCLUDES

$$\frac{\langle\mathcal{A}\rangle\Gamma \vdash_{87} \langle\mathcal{A}\rangle\Gamma \to \langle\overline{\mathcal{A}}\rangle\overline{\Gamma}}{\vdash_{89} \langle\mathcal{A}\rangle\Gamma \to \langle\overline{\mathcal{A}}\rangle\overline{\Gamma}}$$

$$\frac{\begin{array}{cc}\Gamma'(x) \lhd (\alpha, \tau, \{(\mathrm{T}, \sigma)\}) & \langle\mathcal{A}\rangle\Gamma, x, \alpha \vdash_{90} \sigma \to \overline{\sigma} \\ \langle\mathcal{A}\rangle\Gamma \vdash_{87} \langle\mathcal{A}'\rangle\Gamma' \to \langle\overline{\mathcal{A}}'\rangle\overline{\Gamma}' & \overline{\gamma}' \triangleq x \mapsto (\alpha, \tau, \{(\mathrm{T}, \overline{\sigma})\})\end{array}}{\langle\mathcal{A}\rangle\Gamma \vdash_{87} \langle\{x\} \uplus \mathcal{A}'\rangle\Gamma' \to \langle\overline{\mathcal{A}}'\rangle\overline{\Gamma}' \oplus \overline{\gamma}'}$$

$$\langle\mathcal{A}\rangle\Gamma \vdash_{87} \langle\varnothing\rangle\Gamma' \to \langle\varnothing\rangle\Gamma'$$

$$\frac{\langle\mathcal{A}\rangle\Gamma, x, \alpha \vdash_{88} \nu \to \overline{\sigma}_0 \qquad \langle\mathcal{A}\rangle\Gamma, x, \alpha \vdash_{90} \sigma \to \overline{\sigma}}{\langle\mathcal{A}\rangle\Gamma, x, \alpha \vdash_{90} \{\nu\} \uplus \sigma \to \overline{\sigma}_0 \cup \overline{\sigma}} \qquad \langle\mathcal{A}\rangle\Gamma, x, \alpha \vdash_{90} \{\} \to \{\}$$

$$\frac{\langle\mathcal{A}\rangle\Gamma, x, \alpha \vdash_{90} \sigma \to \overline{\sigma} \qquad \langle\mathcal{A}\rangle\Gamma, x, \alpha \vdash_{88} \mathsf{Inter}\,\Sigma \to \{\mathsf{Inter}\,\overline{\Sigma}\}}{\langle\mathcal{A}\rangle\Gamma, x, \alpha \vdash_{88} \mathsf{Inter}\,(\sigma :: \Sigma) \to \{\mathsf{Inter}\,(\overline{\sigma} :: \overline{\Sigma})\}}$$

$$\langle\mathcal{A}\rangle\Gamma, x, \alpha \vdash_{88} \mathsf{Inter}\,[\,] \to \{\mathsf{Inter}\,[\,]\}$$

$$\frac{\begin{array}{ccc}x \neq x' & & x' \in \mathcal{A} \\ \Gamma(x') \lhd (\alpha', \tau', \{(\mathrm{T}', \sigma')\}) & \alpha \sqcap \alpha' \neq \{\} & \langle\mathcal{A}\rangle\Gamma, x', \alpha' \vdash_{90} \sigma' \to \overline{\sigma}'\end{array}}{\langle\mathcal{A}\rangle\Gamma, x, \alpha \vdash_{88} \mathsf{INCLUDES}\,[\,]\,(\mathsf{TRef}\,x')\,\{\} \to \overline{\sigma}'}$$

$$\frac{\langle\mathcal{A}\rangle\Gamma, \text{"INTEGER"}, [\,] \vdash_{90} \sigma \to \overline{\sigma}}{\langle\mathcal{A}\rangle\Gamma, x, \alpha \vdash_{88} \mathsf{SIZE}\,\sigma \to \{\mathsf{SIZE}\,\overline{\sigma}\}}$$

$$\frac{\langle\mathcal{A}\rangle\Gamma, x, \alpha \vdash_{\overline{90}} \sigma \to \overline{\sigma}}{\langle\mathcal{A}\rangle\Gamma, x, \alpha \vdash_{\overline{88}} \mathsf{FROM}\,\sigma \to \{\mathsf{FROM}\,\overline{\sigma}\}}$$

$$\frac{\langle\mathcal{A}\rangle\Gamma, x, \alpha \vdash_{\overline{90}} \sigma \to \overline{\sigma} \qquad \overline{\nu} \triangleq \mathsf{WITH\ COMPONENT}\,\overline{\sigma}}{\langle\mathcal{A}\rangle\Gamma, x, \alpha \vdash_{\overline{88}} \mathsf{WITH\ COMPONENT}\,\sigma \to \{\overline{\nu}\}}$$

$$\frac{\begin{array}{c}\forall j \in [1..p].\langle\mathcal{A}\rangle\Gamma, x, \alpha \vdash_{\overline{90}} \sigma_j \to \overline{\sigma}_j \\ \overline{\nu} \triangleq \mathsf{WITH\ COMPONENTS}\ (m, [(l_j, \overline{\sigma}_j, \hat{\pi}_j)]_{1\leqslant j\leqslant p})\end{array}}{\langle\mathcal{A}\rangle\Gamma, x, \alpha \vdash_{\overline{88}} \mathsf{WITH\ COMPONENTS}\ (m, [(l_j, \sigma_j, \hat{\pi}_j)]_{1\leqslant j\leqslant p}) \to \{\overline{\nu}\}}$$

$$\frac{\nu \lhd \mathsf{Value}\ \_ \mid \_\_..\_\_}{\langle\mathcal{A}\rangle\Gamma, x, \alpha \vdash_{\overline{88}} \nu \to \{\nu\}}$$

9.2 Forme disjonctive

```
let merge = function
  (Inter Σ̄0, Inter Σ̄1) → Σ̄0 @ Σ̄1
| (Inter Σ̄0, ν̄1) → Σ̄0 @ [{ν̄1}]
| (ν̄0, Inter Σ̄1) → {ν̄0} :: Σ̄1
| (ν̄0, ν̄1) → [{ν̄0}; {ν̄1}]

let rec distribute_right = function
  (ν̄0, {ν̄1} ⊎ σ̄1) → {Inter (merge (ν̄0, ν̄1))} ∪ (distribute_right (ν̄0, σ̄1))
| _ → {}

let rec distribute_left = function
  ({ν̄0} ⊎ σ̄0, σ̄1) → (distribute_right (ν̄0, σ̄1)) ∪ (distribute_left (σ̄0, σ̄1))
| _ → {}

let rec disjonctive_intersection = function
  σ :: Σ → let σ̄0 = disjonctive_constraint σ
           in (match disjonctive_intersection Σ with
                 {} → σ̄0
               | σ̄1 → distribute_left (σ̄0, σ̄1))
| [] → {}

and disjonctive_constraint = function
  {ν} ⊎ σ → let σ̄0 = disjonctive_base_and_intersection ν in
              let σ̄1 = disjonctive_constraint σ
            in σ̄0 ∪ σ̄1
```

$|\ \{\} \to \{\}$

and disjonctive_base_and_intersection = **function**
  Inter $\Sigma \to$ disjonctive_intersection Σ
| SIZE $\sigma \to$ **let** $\{\overline{\nu}_i\}_{1 \leqslant i \leqslant n}$ = disjonctive_constraint σ
  in $\bigcup_{i=1}^{n}$ {SIZE $\{\overline{\nu}_i\}$}
| FROM $\sigma \to$ {FROM (disjonctive_constraint σ)}
| INCLUDES τ T $\sigma \to$ {INCLUDES τ T (disjonctive_constraint σ)}
| WITH COMPONENT $\sigma \to$
  {WITH COMPONENT (disjonctive_constraint σ)}
| WITH COMPONENTS $(m, [(l_j, \sigma_j, \hat{\pi}_j)]_{1 \leqslant j \leqslant p}) \to$
{WITH COMPONENTS $(m, [(l_j, \text{disjonctive\_constraint}\,(\sigma_j), \hat{\pi}_j)]_{1 \leqslant j \leqslant p})$}
| $\nu \to \{\nu\}$

9.3 Distribution des contraintes disjonctives

$$\frac{\Gamma(x) \lhd (\alpha, \tau, \{(\mathrm{T}, \{\})\}) \qquad \vdash_{\overline{62}} \langle \mathcal{A} \rangle \Gamma \to \langle \overline{\mathcal{A}} \rangle \overline{\Gamma}}{\vdash_{\overline{62}} \langle \{x\} \uplus \mathcal{A} \rangle \Gamma \to \langle \overline{\mathcal{A}} \rangle \overline{\Gamma} \oplus x \mapsto \Gamma(x)}$$

$$\frac{\begin{array}{c}\Gamma(x) \lhd (\alpha, \tau, \{(\mathrm{T}, \sigma)\}) \\ \sigma \neq \{\} \qquad \vdash_{\overline{62}} \langle \mathcal{A} \rangle \Gamma \to \langle \overline{\mathcal{A}} \rangle \overline{\Gamma} \qquad \{\overline{\nu}_i\}_{1 \leqslant i \leqslant n} \triangleq \text{disjonctive\_constraint}\,(\sigma) \\ \overline{\gamma} \triangleq x \mapsto (\alpha, \tau, \{(\mathrm{T}, \{\overline{\nu}_i\})\}_{1 \leqslant i \leqslant n})\end{array}}{\vdash_{\overline{62}} \langle \{x\} \uplus \mathcal{A} \rangle \Gamma \to \langle \overline{\mathcal{A}} \rangle \overline{\Gamma} \oplus \overline{\gamma}}$$

$$\vdash_{\overline{62}} \langle \mathcal{A} \rangle \Gamma \to \langle \mathcal{A} \rangle \Gamma$$

9.4 Recouplage valeurs/contraintes

$$\frac{\Gamma(x) \lhd (\alpha, \tau, \{(\mathrm{T}, \{\})\}) \qquad \langle \mathcal{B} \rangle \Delta \vdash_{\overline{43}} \langle \mathcal{A} \rangle \Gamma \to \langle \overline{\mathcal{A}} \rangle \overline{\Gamma}}{\langle \mathcal{B} \rangle \Delta \vdash_{\overline{43}} \langle \{x\} \uplus \mathcal{A} \rangle \Gamma \to \langle \overline{\mathcal{A}} \rangle \overline{\Gamma} \oplus x \mapsto \Gamma(x)}$$

$$\frac{\begin{array}{c}\Gamma(x) \lhd (\alpha, \tau, \{(\mathrm{T}, \{\nu_i\})\}_{1 \leqslant i \leqslant n}) \qquad \forall i \in [1..n].\langle \mathcal{B} \rangle \Delta \vdash_{\overline{56}} \nu_i \to \overline{\nu}_i \\ \langle \mathcal{B} \rangle \Delta \vdash_{\overline{43}} \langle \mathcal{A} \rangle \Gamma \to \langle \overline{\mathcal{A}} \rangle \overline{\Gamma} \qquad \overline{\gamma} \triangleq x \mapsto (\alpha, \tau, \{(\mathrm{T}, \{\overline{\nu}_i\})\}_{1 \leqslant i \leqslant n})\end{array}}{\langle \mathcal{B} \rangle \Delta \vdash_{\overline{43}} \langle \{x\} \uplus \mathcal{A} \rangle \Gamma \to \langle \overline{\mathcal{A}} \rangle \overline{\Gamma} \oplus \overline{\gamma}}$$

$$\langle \mathcal{B} \rangle \Delta \vdash_{\overline{43}} \langle \varnothing \rangle \Gamma \to \langle \varnothing \rangle \Gamma$$

$$\frac{\langle \mathcal{B} \rangle \Delta \vdash_{\overline{56}} \nu' \to \overline{\nu}' \qquad \langle \mathcal{B} \rangle \Delta \vdash_{\overline{56}} \mathsf{Inter}\,\Sigma' \to \mathsf{Inter}\,\overline{\Sigma}'}{\langle \mathcal{B} \rangle \Delta \vdash_{\overline{56}} \mathsf{Inter}\,(\{\nu'\} :: \Sigma') \to \mathsf{Inter}\,(\{\overline{\nu}'\} :: \overline{\Sigma}')}$$

$$\langle\mathcal{B}\rangle\Delta \vdash_{\overline{5}6} \mathsf{Inter}\,[\,] \to \mathsf{Inter}\,[\,] \qquad \frac{y \in \mathcal{B} \qquad \Delta(y) \lhd ([\,], \mathsf{TRef}\ \_, \{\}, v)}{\langle\mathcal{B}\rangle\Delta \vdash_{\overline{5}6} \mathsf{Value}\,(\mathsf{VRef}\,y) \to \mathsf{Value}\,(v)}$$

$$\frac{\begin{array}{c} y_0 \in \mathcal{B} \wedge \Delta(y_0) \lhd ([\,], \mathsf{TRef}\ \_, \{\}, v_0) \\ y_1 \in \mathcal{B} \wedge \Delta(y_1) \lhd ([\,], \mathsf{TRef}\ \_, \{\}, v_1) \qquad \overline{\nu} \triangleq (\mathsf{Val}\,v_0)\,b_0..b_1\,(\mathsf{Val}\,v_1) \end{array}}{\langle\mathcal{B}\rangle\Delta \vdash_{\overline{5}6} (\mathsf{Val}\,(\mathsf{VRef}\,y_0))\,b_0..b_1\,(\mathsf{Val}\,(\mathsf{VRef}\,y_1)) \to \overline{\nu}}$$

$$\frac{y_0 \in \mathcal{B} \qquad \Delta(y_0) \lhd ([\,], \mathsf{TRef}\ \_, \{\}, v_0) \qquad \overline{\nu} \triangleq (\mathsf{Val}\,v_0)\,b_0..b_1\,e_1}{\langle\mathcal{B}\rangle\Delta \vdash_{\overline{5}6} (\mathsf{Val}\,(\mathsf{VRef}\,y_0))\,b_0..b_1\,e_1}$$

$$\frac{y_1 \in \Delta \qquad \Delta(y_1) \lhd ([\,], \mathsf{TRef}\ \_, \{\}, v_1) \qquad \overline{\nu} \triangleq e_0\,b_0..b_1\,(\mathsf{Val}\,v_1)}{\langle\mathcal{B}\rangle\Delta \vdash_{\overline{5}6} e_0\,b_0..b_1\,(\mathsf{Val}\,(\mathsf{VRef}\,y_1)) \to \overline{\nu}}$$

$$\frac{\langle\mathcal{B}\rangle\Delta \vdash_{\overline{5}6} \nu \to \overline{\nu}}{\langle\mathcal{B}\rangle\Delta \vdash_{\overline{5}6} \mathsf{SIZE}\,\{\nu\} \to \mathsf{SIZE}\,\{\overline{\nu}\}} \qquad \frac{\langle\mathcal{B}\rangle\Delta \vdash_{\overline{5}7} \sigma \to \overline{\sigma}}{\langle\mathcal{B}\rangle\Delta \vdash_{\overline{5}6} \mathsf{FROM}\,\sigma \to \mathsf{FROM}\,\overline{\sigma}}$$

$$\frac{\langle\mathcal{B}\rangle\Delta \vdash_{\overline{5}7} \sigma \to \overline{\sigma} \qquad \overline{\nu} \triangleq \mathsf{WITH\ COMPONENT}\ \overline{\sigma}}{\langle\mathcal{B}\rangle\Delta \vdash_{\overline{5}6} \mathsf{WITH\ COMPONENT}\ \sigma \to \overline{\nu}}$$

$$\frac{\begin{array}{c} \forall j \in [1..p].\langle\mathcal{B}\rangle\Delta \vdash_{\overline{5}7} \sigma_j \to \overline{\sigma}_j \\ \overline{\nu} \triangleq \mathsf{WITH\ COMPONENTS}\ (m, [(l_j, \overline{\sigma}_j, \hat{\pi}_j)]_{1 \leqslant j \leqslant p}) \end{array}}{\langle\mathcal{B}\rangle\Delta \vdash_{\overline{5}6} \mathsf{WITH\ COMPONENTS}\ (m, [(l_j, \sigma_j, \hat{\pi}_j)]_{1 \leqslant j \leqslant p}) \to \overline{\nu}}$$

$$\frac{\langle\mathcal{B}\rangle\Delta \vdash_{\overline{5}6} \nu \to \overline{\nu} \qquad \langle\mathcal{B}\rangle\Delta \vdash_{\overline{5}7} \sigma \to \overline{\sigma}}{\langle\mathcal{B}\rangle\Delta \vdash_{\overline{5}7} \{\nu\} \uplus \sigma \to \{\overline{\nu}\} \cup \overline{\sigma}} \qquad \langle\mathcal{B}\rangle\Delta \vdash_{\overline{5}7} \{\} \to \{\}$$

9.5 Contraintes internes

9.5.1 Réduction partielle des contraintes internes

$$\frac{\begin{array}{c} \Gamma(x) \lhd (\alpha, \tau, \{(\mathrm{T}_k, \{\nu_k\})\}_{1 \leqslant k \leqslant q}) \qquad \forall k \in [1..q].x \vdash_{\overline{5}8} (\mathrm{T}_k, \nu_k) \to (\overline{\mathrm{T}}_k, \overline{\sigma}_k) \\ \vdash_{\overline{1}4} \langle\mathcal{A}\rangle\Gamma \to \langle\overline{\mathcal{A}}\rangle\overline{\Gamma} \qquad \overline{\gamma} \triangleq x \mapsto (\alpha, \tau, \{(\overline{\mathrm{T}}_k, \overline{\sigma}_k)\}_{1 \leqslant k \leqslant q}) \end{array}}{\vdash_{\overline{1}4} \langle\{x\} \uplus \mathcal{A}\rangle\Gamma \to \langle\overline{\mathcal{A}}\rangle\overline{\Gamma} \oplus \overline{\gamma}}$$

$$\vdash_{\overline{1}4} \langle\varnothing\rangle\Gamma \to \langle\varnothing\rangle\Gamma \qquad \frac{x, \mathrm{T} \vdash_{\overline{8}6} \nu \to \nu_0 \qquad \vdash_{\overline{1}3} (\mathrm{T}, \nu_0) \to r}{x, \vdash_{\overline{5}8} (\mathrm{T}, \nu) \to r}$$

Contraintes internes des contraintes par valeurs

$$\frac{x, \mathrm{T} \vdash_{86} \nu' \to \overline{\nu}'_0 \qquad x, \mathrm{T} \vdash_{86} \mathsf{Inter}\,\Sigma' \to \overline{\nu}'_1 \qquad \overline{\nu} \triangleq \mathsf{Inter}\,(\mathrm{merge}\,(\overline{\nu}'_0, \overline{\nu}'_1))}{x, \mathrm{T} \vdash_{86} \mathsf{Inter}\,(\{\nu'\} :: \Sigma') \to \overline{\nu}}$$

$$\frac{\begin{array}{c}\mathrm{T} \lhd (\mathsf{SEQUENCE\ OF} \mid \mathsf{SET\ OF})\ \_\ \_\ \_ \qquad \overline{\nu}_0 \triangleq \mathsf{SIZE}\,\{\mathsf{Value}\,(\mathsf{Int}\,n)\} \\ \overline{\nu}_1 \triangleq \mathsf{WITH\ COMPONENT}\,\{\mathsf{Value}\,(v_i)\}_{1 \leqslant i \leqslant n} \\ \overline{\nu} \triangleq \mathsf{Inter}\,[\{\nu\}; \{\overline{\nu}_0\}; \{\overline{\nu}_1\}]\end{array}}{x, \mathrm{T} \vdash_{86} \mathsf{Value}\,\mathbf{\{}[(\mathsf{None}, v_i)]_{1 \leqslant i \leqslant n}\mathbf{\}}\,\mathbf{as}\,\nu \to \overline{\nu}}$$

$$\frac{\begin{array}{c}\mathrm{T} \lhd (\mathsf{SEQUENCE\ OF} \mid \mathsf{SET\ OF})\ \_\ \_\ \_ \\ \overline{\nu}_0 \triangleq \mathsf{SIZE}\,\{\mathsf{Value}\,(\mathsf{Int}\,0)\} \qquad \overline{\nu} \triangleq \mathsf{Inter}\,[\{\nu\}; \{\overline{\nu}_0\}]\end{array}}{x, \mathrm{T} \vdash_{86} \mathsf{Value}\,\mathbf{\{]\}}\,\mathbf{as}\,\nu \to \overline{\nu}}$$

$$\frac{\begin{array}{c}\mathrm{T} \lhd \mathsf{CHOICE}\ \ \_ \qquad \mathcal{K} \triangleq [(x', \{\mathsf{Value}\,(v')\}, \mathsf{Some\,PRESENT})] \\ \overline{\nu}_0 \triangleq \mathsf{WITH\ COMPONENTS}\,\mathbf{\{}\mathsf{Full}\ \mathcal{K}\mathbf{\}} \qquad \overline{\nu} \triangleq \mathsf{Inter}\,[\{\nu\}; \{\overline{\nu}_0\}]\end{array}}{x, \mathrm{T} \vdash_{86} \mathsf{Value}\,(x' : v')\,\mathbf{as}\,\nu \to \overline{\nu}}$$

$$\frac{x, \mathsf{SEQUENCE}\,\Phi \vdash_{86} \nu \to \overline{\nu}}{x, \mathsf{SET}\,\Phi \vdash_{86} \nu \to \overline{\nu}}$$

$$\frac{\begin{array}{c}x \neq \text{"REAL"} \\ \mathrm{T} \lhd \mathsf{SEQUENCE}\,(\varphi' :: \Phi') \qquad \varphi' \lhd \mathsf{Field}\,(l', [\,], \mathsf{TRef}\,x, \{\}, s') \\ V \lhd [(\mathsf{Some}\,l', v')] \sqcup V' \qquad x, \mathsf{SEQUENCE}\,\Phi' \vdash_{86} \mathsf{Value}\,\mathbf{\{}V'\mathbf{\}} \to \overline{\nu}' \\ \overline{\nu}' \lhd \mathsf{Inter}\,[\{\overline{\nu}'_0\}; \{\overline{\nu}'_1\}] \qquad \overline{\nu}'_1 \lhd \mathsf{WITH\ COMPONENTS}\,\mathbf{\{}\mathsf{Full}\ \mathcal{K}'\mathbf{\}} \\ \mathcal{K} \triangleq (l', \{\mathsf{Value}\,(v')\}, \mathsf{Some\,PRESENT}) :: \mathcal{K}' \\ \overline{\nu}_1 \triangleq \mathsf{WITH\ COMPONENTS}\,\mathbf{\{}\mathsf{Full}\ \mathcal{K}\mathbf{\}} \qquad \overline{\nu} \triangleq \mathsf{Inter}\,[\{\nu\}; \{\overline{\nu}_1\}]\end{array}}{x, \mathrm{T} \vdash_{86} \mathsf{Value}\,\mathbf{\{}V\mathbf{\}}\,\mathbf{as}\,\nu \to \overline{\nu}}$$

$$\frac{\begin{array}{c}x \neq \text{"REAL"} \\ \mathrm{T} \lhd \mathsf{SEQUENCE}\,(\varphi' :: \Phi')\ \ \varphi' \lhd \mathsf{Field}\,(l', [\,], \mathsf{TRef}\,x, \{\}, s') \\ \forall (\mathsf{Some}\,x', v') \in V. x' \neq l' \qquad x, \mathsf{SEQUENCE}\,\Phi' \vdash_{86} \nu \to \overline{\nu}' \\ \overline{\nu}' \lhd \mathsf{Inter}\,[\{\overline{\nu}'_0\}; \{\overline{\nu}'_1\}] \qquad \overline{\nu}'_1 \lhd \mathsf{WITH\ COMPONENTS}\,\mathbf{\{}\mathsf{Full}\ \mathcal{K}'\mathbf{\}} \\ \mathcal{K} \triangleq (l', \{\}, \mathsf{Some\,ABSENT}) :: \mathcal{K}' \\ \overline{\nu}_1 \triangleq \mathsf{WITH\ COMPONENTS}\,\mathbf{\{}\mathsf{Full}\ \mathcal{K}\mathbf{\}} \qquad \overline{\nu} \triangleq \mathsf{Inter}\,[\{\nu\}; \{\overline{\nu}_1\}]\end{array}}{x, \mathrm{T} \vdash_{86} \mathsf{Value}\,\mathbf{\{}V\mathbf{\}}\,\mathbf{as}\,\nu \to \overline{\nu}}$$

$$\frac{\begin{array}{c}\mathrm{T} \lhd \mathsf{SEQUENCE}\,[\,] \\ \overline{\nu}_1 \triangleq \mathsf{WITH\ COMPONENTS}\,\mathbf{\{}\mathsf{Full}\ [\,]\mathbf{\}} \qquad \overline{\nu} \triangleq \mathsf{Inter}\,[\{\nu\}; \{\overline{\nu}_1\}]\end{array}}{x, \mathrm{T} \vdash_{86} \mathsf{Value}\,\mathbf{\{]\}}\,\mathbf{as}\,\nu \to \overline{\nu}}$$

$$\text{"REAL"}, \mathrm{T} \vdash_{\overline{86}} \nu \to \nu$$

$$\frac{\mathrm{T} \not\lhd \mathsf{SEQUENCE\ OF\ \_\_\_} \mid \mathsf{SET\ OF\ \_\_\_} \mid \mathsf{CHOICE\ \_} \qquad \mathrm{T} \not\lhd \mathsf{SET\ \_} \mid \mathsf{SEQUENCE\ \_}}{x, \mathrm{T} \vdash_{\overline{86}} \nu \to \nu}$$

Réduction des contraintes internes

$$\frac{\vdash_{\overline{13}} (\mathrm{T}, \nu') \to (\overline{\mathrm{T}}_0, \overline{\sigma}_0) \qquad \vdash_{\overline{13}} (\overline{\mathrm{T}}_0, \mathsf{Inter}\,\Sigma') \to (\overline{\mathrm{T}}, \overline{\sigma}_1) \qquad \overline{\Sigma} \triangleq \mathrm{list\_of\_set}\,(\overline{\sigma}_0 \cup \overline{\sigma}_1) \qquad \vdash_{\overline{16}} \overline{\Sigma} \to \overline{\nu}}{\vdash_{\overline{13}} (\mathrm{T}, \mathsf{Inter}\,(\{\nu'\} :: \Sigma')) \to (\overline{\mathrm{T}}, \{\overline{\nu}\})}$$

$$\vdash_{\overline{13}} (\mathrm{T}, \mathsf{Inter}\,[\,]) \to (\mathrm{T}, \{\})$$

$$\frac{\mathrm{T} \lhd \mathsf{SET\ OF}\ [\,]\ (\mathsf{TRef}\,x)\,\sigma' \qquad \overline{\mathrm{T}} \triangleq \mathsf{SET\ OF}\ [\,]\ (\mathsf{TRef}\,x)\,(\mathsf{Inter}\,[\sigma'; \sigma])}{\vdash_{\overline{13}} (\mathrm{T}, \mathsf{WITH\ COMPONENT}\ \sigma) \to (\overline{\mathrm{T}}, \{\})}$$

$$\frac{\mathrm{T} \lhd \mathsf{SEQUENCE\ OF}\ [\,]\ (\mathsf{TRef}\,x)\,\sigma' \qquad \overline{\mathrm{T}} \triangleq \mathsf{SEQUENCE\ OF}\ [\,]\ (\mathsf{TRef}\,x)\,(\mathsf{Inter}\,[\sigma'; \sigma])}{\vdash_{\overline{13}} (\mathrm{T}, \mathsf{WITH\ COMPONENT}\ \sigma) \to (\overline{\mathrm{T}}, \{\})}$$

$$\frac{[\,], m \vdash_{\overline{12}} (\mathrm{T}, \mathcal{K}) \to \overline{\mathrm{T}}}{\vdash_{\overline{13}} (\mathrm{T}, \mathsf{WITH\ COMPONENTS}\ \{m\ \mathcal{K}\}) \to (\overline{\mathrm{T}}, \{\})}$$

$$\frac{\nu \not\lhd \mathsf{Inter\ \_} \mid \mathsf{WITH\ COMPONENT\ \_} \mid \mathsf{WITH\ COMPONENTS\ \_}}{\vdash_{\overline{13}} (\mathrm{T}, \nu) \to (\mathrm{T}, \nu)}$$

$$\vdash_{\overline{16}} [\,] \to \{\} \qquad \vdash_{\overline{16}} [\overline{\sigma}] \to \overline{\sigma} \qquad \frac{\overline{\Sigma} \lhd \_ :: \_ :: \_}{\vdash_{\overline{16}} \overline{\Sigma} \to \{\mathsf{Inter}\,\overline{\Sigma}\}}$$

$$\frac{f' \lhd (l', \tau', \mathrm{T}', \{\}, s') \qquad k' \lhd (l', \sigma, \mathsf{None}) \qquad \overline{f}' \triangleq (l', \tau', \mathrm{T}', \sigma, s') \qquad \overline{\mathcal{F}} \mathbin{@} [\overline{f}'], m \vdash_{\overline{12}} \mathsf{CHOICE}\,(\mathcal{F}') \parallel \mathcal{K}' \to \overline{\mathrm{T}}}{\overline{\mathcal{F}}, m \vdash_{\overline{12}} \mathsf{CHOICE}\,([f'] \sqcup \mathcal{F}') \parallel [k'] \sqcup \mathcal{K}' \to \overline{\mathrm{T}}}$$

$$\frac{(l', \tau', \mathrm{T}', \{\}, s') \in \mathcal{F} \qquad k' \lhd (l', \sigma, \mathsf{Some\ PRESENT}) \qquad (\_, \_, \mathsf{Some\ PRESENT}) \notin \mathcal{K}' \qquad \overline{f}' \triangleq (l', \tau', \mathrm{T}', \sigma, s')}{\overline{\mathcal{F}}, m \vdash_{\overline{12}} \mathsf{CHOICE}\,(\mathcal{F}) \parallel [k'] \sqcup \mathcal{K}' \to \mathsf{CHOICE}\,[\overline{f}']}$$

$$\frac{f' \lhd (l', \tau', \mathrm{T}', \sigma', s') \qquad k' \lhd (l', \_, \mathsf{Some\ ABSENT}) \qquad \overline{\mathcal{F}} \not\triangleleft [\,] \ \vee\ \mathcal{F}' \not\triangleleft [\,] \qquad \overline{\mathcal{F}}, m \vdash_{\overline{1}2} \mathsf{CHOICE}\,(\mathcal{F}') \parallel \mathcal{K}' \to \overline{\mathrm{T}}}{\overline{\mathcal{F}}, m \vdash_{\overline{1}2} \mathsf{CHOICE}\,([f'] \sqcup \mathcal{F}') \parallel [k'] \sqcup \mathcal{K}' \to \overline{\mathrm{T}}}$$

$$\overline{\mathcal{F}}, \mathsf{Partial} \vdash_{\overline{1}2} \mathsf{CHOICE}\,(\mathcal{F}) \parallel [\,] \to \mathsf{CHOICE}\,(\overline{\mathcal{F}} \mathbin{@} \mathcal{F})$$

$$\overline{\mathcal{F}}, \mathsf{Full} \vdash_{\overline{1}2} \mathsf{CHOICE}\ \ \_ \parallel [\,] \to \mathsf{CHOICE}\,\overline{\mathcal{F}}$$

$$\frac{f' \lhd (l', \tau', \mathrm{T}', \sigma', s') \qquad (l', \_, \_) \notin \mathcal{K} \qquad \overline{\mathcal{F}} \mathbin{@} [f'], m \vdash_{\overline{1}2} \mathsf{CHOICE}\,(\mathcal{F}') \parallel \mathcal{K} \to \overline{\mathrm{T}}}{\overline{\mathcal{F}}, m \vdash_{\overline{1}2} \mathsf{CHOICE}\,([f'] \sqcup \mathcal{F}') \parallel \mathcal{K} \to \overline{\mathrm{T}}}$$

$$\frac{\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \qquad s' \lhd \mathsf{Some\ OPTIONAL} \qquad k' \lhd (l', \sigma, \mathsf{Some\ ABSENT}) \qquad \overline{\mathcal{F}}, m \vdash_{\overline{1}2} \mathsf{SEQUENCE}\,\Phi' \parallel \mathcal{K}' \to \overline{\mathrm{T}}}{\overline{\mathcal{F}}, m \vdash_{\overline{1}2} \mathsf{SEQUENCE}\,(\varphi' :: \Phi') \parallel k' :: \mathcal{K}' \to \overline{\mathrm{T}}}$$

$$\frac{\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \{\}, s') \qquad k' \lhd (l', \sigma, \mathsf{None}) \qquad \overline{f}' \triangleq (l', \tau', \mathrm{T}', \sigma, s') \qquad \overline{\mathcal{F}} \mathbin{@} [\overline{f}'], \mathsf{Partial} \vdash_{\overline{1}2} \mathsf{SEQUENCE}\,(\Phi') \parallel \mathcal{K}' \to \overline{\mathrm{T}}}{\overline{\mathcal{F}}, \mathsf{Partial} \vdash_{\overline{1}2} \mathsf{SEQUENCE}\,(\varphi' :: \Phi') \parallel k' :: \mathcal{K}' \to \overline{\mathrm{T}}}$$

$$\frac{\mathrm{T} \lhd \mathsf{SEQUENCE}\,(\varphi' :: \Phi') \qquad \varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \{\}, s') \qquad s' \lhd \mathsf{Some\,OPTIONAL} \qquad k' \lhd (l', \sigma, \mathsf{None}) \qquad \overline{\mathcal{K}} \triangleq (l', \sigma, \mathsf{Some\ PRESENT}) :: \mathcal{K}' \qquad \overline{\mathcal{F}}, \mathsf{Full} \vdash_{\overline{1}2} \mathrm{T} \parallel \overline{\mathcal{K}} \to \overline{\mathrm{T}}}{\overline{\mathcal{F}}, \mathsf{Full} \vdash_{\overline{1}2} \mathrm{T} \parallel k' :: \mathcal{K}' \to \overline{\mathrm{T}}}$$

$$\frac{\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \{\}, s') \qquad s' \lhd \mathsf{Some\,OPTIONAL} \qquad k' \lhd (l', \sigma, \mathsf{Some\,OPTIONAL}) \qquad \overline{f}' \triangleq (l', \tau', \mathrm{T}', \sigma, s') \qquad \overline{\mathcal{F}} \mathbin{@} [\overline{f}'], m \vdash_{\overline{1}2} \mathsf{SEQUENCE}\,(\Phi') \parallel \mathcal{K}' \to \overline{\mathrm{T}}}{\overline{\mathcal{F}}, m \vdash_{\overline{1}2} \mathsf{SEQUENCE}\,(\varphi' :: \Phi') \parallel k' :: \mathcal{K}' \to \overline{\mathrm{T}}}$$

$$\frac{\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \{\}, s') \qquad s' \lhd \mathsf{Some\,OPTIONAL} \qquad k' \lhd (l', \sigma, \mathsf{Some\ PRESENT}) \qquad \overline{f}' \triangleq (l', \tau', \mathrm{T}', \sigma, \mathsf{None}) \qquad \overline{\mathcal{F}} \mathbin{@} [\overline{f}'], m \vdash_{\overline{1}2} \mathsf{SEQUENCE}\,(\Phi') \parallel \mathcal{K}' \to \overline{\mathrm{T}}}{\overline{\mathcal{F}}, m \vdash_{\overline{1}2} \mathsf{SEQUENCE}\,(\varphi' :: \Phi') \parallel k' :: \mathcal{K}' \to \overline{\mathrm{T}}}$$

$$\frac{\varphi' \lhd \mathsf{Field}\,(f') \qquad f' \lhd (l', \tau', \mathrm{T}', \sigma', s') \qquad \mathcal{K} \lhd (l, \sigma, \hat{\pi}) :: \mathcal{K}' \qquad l \neq l' \qquad \overline{\mathcal{F}} \mathbin{@} [f'], m \vdash_{\overline{1}2} \mathsf{SEQUENCE}\,(\Phi') \parallel \mathcal{K} \to \overline{\mathrm{T}}}{\overline{\mathcal{F}}, m \vdash_{\overline{1}2} \mathsf{SEQUENCE}\,(\varphi' :: \Phi') \parallel \mathcal{K} \to \overline{\mathrm{T}}}$$

$$\frac{\overline{\Phi} \triangleq \mathrm{List.map}\ \mathsf{Field}\ \overline{\mathcal{F}}}{\overline{\mathcal{F}}, \mathsf{Partial} \vdash_{\overline{12}} \mathsf{SEQUENCE}\,(\Phi) \parallel [\,] \to \mathsf{SEQUENCE}\,(\overline{\Phi} \mathbin{@} \Phi)}$$

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\,(l', \tau', \mathrm{T}', \sigma', s') \\ s' \lhd \mathsf{Some}\,\mathsf{OPTIONAL} \qquad \overline{\mathcal{K}} \triangleq [(l', \{\}, \mathsf{Some}\,\mathsf{ABSENT})] \\ \overline{\mathcal{F}}, \mathsf{Full} \vdash_{\overline{12}} \mathsf{SEQUENCE}\,([\varphi'] \sqcup \Phi') \parallel \overline{\mathcal{K}} \to \overline{\mathrm{T}}\end{array}}{\overline{\mathcal{F}}, \mathsf{Full} \vdash_{\overline{12}} \mathsf{SEQUENCE}\,([\varphi'] \sqcup \Phi') \mid [\,] \to \overline{\mathrm{T}}}$$

$$\frac{\begin{array}{c}\varphi' \lhd \mathsf{Field}\,(f') \qquad f' \lhd (l', \tau', \mathrm{T}', \sigma', s') \\ s' \not\lhd \mathsf{Some}\,\mathsf{OPTIONAL} \qquad \overline{\mathcal{F}} \mathbin{@} [f'], \mathsf{Full} \vdash_{\overline{12}} \mathsf{SEQUENCE}\,(\Phi') \mid [\,] \to \overline{\mathrm{T}}\end{array}}{\overline{\mathcal{F}}, \mathsf{Full} \vdash_{\overline{12}} \mathsf{SEQUENCE}\,([\varphi'] \sqcup \Phi') \mid [\,] \to \overline{\mathrm{T}}}$$

$$\frac{\overline{\Phi} \triangleq \mathrm{List.map}\ \mathsf{Field}\ \overline{\mathcal{F}}}{\overline{\mathcal{F}}, \mathsf{Full} \vdash_{\overline{12}} \mathsf{SEQUENCE}\,[\,] \parallel [\,] \to \mathsf{SEQUENCE}\,\overline{\Phi}}$$

$$\frac{\overline{\mathcal{F}}, m \vdash_{\overline{12}} \mathsf{SEQUENCE}\,(\Phi) \parallel \pi\,(\mathcal{K}) \to \overline{\mathrm{T}} \qquad \pi \textit{ est une permutation}}{\overline{\mathcal{F}}, m \vdash_{\overline{12}} \mathsf{SET}\,(\Phi) \parallel \mathcal{K} \to \overline{\mathrm{T}}}$$

9.5.2 Réduction complète des contraintes internes

$$\frac{\vdash_{\overline{19}} \langle\mathcal{A}\rangle\Gamma \to \langle\mathcal{A}\rangle\Gamma}{\vdash_{\overline{20}} \langle\mathcal{A}\rangle\Gamma \to \langle\mathcal{A}\rangle\Gamma}$$

$$\frac{\vdash_{\overline{19}} \langle\mathcal{A}\rangle\Gamma \to \langle\mathcal{A}_0\rangle\Gamma_0 \qquad \langle\mathcal{A}\rangle\Gamma \neq \langle\mathcal{A}_0\rangle\Gamma_0 \qquad \vdash_{\overline{20}} \langle\mathcal{A}_0\rangle\Gamma_0 \to \langle\mathcal{A}_1\rangle\Gamma_1}{\vdash_{\overline{20}} \langle\mathcal{A}\rangle\Gamma \to \langle\mathcal{A}_1\rangle\Gamma_1}$$

$$\frac{\begin{array}{ccc} & \vdash_{\overline{14}} \langle\mathcal{A}\rangle\Gamma \to \langle\mathcal{A}_0\rangle\Gamma_0 & \\ \vdash_{\overline{65}} \langle\mathcal{A}_0\rangle\Gamma_0 \to \langle\mathcal{A}_1\rangle\Gamma_1 & \vdash_{\overline{68}} \langle\mathcal{A}_1\rangle\Gamma_1 \to \langle\mathcal{A}_2\rangle\Gamma_2 & \vdash_{\overline{62}} \langle\mathcal{A}_2\rangle\Gamma_2 \to \langle\overline{\mathcal{A}}\rangle\overline{\Gamma}\end{array}}{\vdash_{\overline{19}} \langle\mathcal{A}\rangle\Gamma \to \langle\overline{\mathcal{A}}\rangle\overline{\Gamma}}$$

9.6 Contraintes d'intervalle

9.6.1 Normalisation des bornes finies réelles

$$\frac{\vdash_{\overline{97}} e_0 \to \overline{e}_0 \qquad \vdash_{\overline{97}} e_1 \to \overline{e}_1}{\vdash_{\overline{50}} e_0\, b_0 \mathinner{..} b_1\, e_1 \to \overline{e}_0\, b_0 \mathinner{..} b_1\, \overline{e}_1} \qquad \frac{\nu \not\lhd \_\_\mathinner{..}\_\_}{\vdash_{\overline{50}} \nu \to \nu}$$

$$\frac{\overline{v} \triangleq \mathrm{Real.normalize}\,(v)}{\vdash_{\overline{97}} \mathsf{Val}\,(v) \to \mathsf{Val}\,(\overline{v})} \qquad \frac{e \not\lhd \mathsf{Val}\ \_}{\vdash_{\overline{97}} e \to e}$$

9.6.2 Normalisation des bornes MAX et MIN

$$\frac{x, \mathrm{T} \vdash_{\overline{61}} e_0 \mid \Sigma \to \overline{e}_0 \qquad x, \mathrm{T} \vdash_{\overline{61}} e_1 \mid \Sigma \to \overline{e}_1}{x, \mathrm{T} \vdash_{\overline{45}} e_0 b_0 \,..\, b_1 e_1 \mid \Sigma \to \overline{e}_0 b_0 \,..\, b_1 \overline{e}_1} \qquad \frac{x, \mathrm{T} \vdash_{\overline{59}} \Sigma \to \min}{x, \mathrm{T} \vdash_{\overline{61}} \mathsf{MIN} \mid \Sigma \to \min}$$

$$\frac{x, \mathrm{T} \vdash_{\overline{60}} \Sigma \to \max}{x, \mathrm{T} \vdash_{\overline{61}} \mathsf{MAX} \mid \Sigma \to \max} \qquad \frac{e \not\lhd \mathsf{MIN} \mid \mathsf{MAX}}{x, \mathrm{T} \vdash_{\overline{61}} e \mid \Sigma \to e}$$

Borne inférieure du premier intervalle dans une liste de contraintes

$$x, \mathrm{T} \vdash_{\overline{59}} \{\mathsf{Value}\,(v)\} :: \Sigma \to \mathsf{Val}\,(v) \qquad x, \mathrm{T} \vdash_{\overline{59}} \{e_0 b_0 \,..\, b_1 e_1\} :: \Sigma \to e_0$$

$$x, \mathsf{INTEGER} \;\; \_ \vdash_{\overline{59}} [\,] \to \mathsf{MIN}$$

$$\text{"REAL"}, \mathrm{T} \vdash_{\overline{59}} [\,] \to \mathsf{Val}\,(\mathsf{MINUS\text{-}INFINITY})$$

$$x, \mathsf{CharString}\,(\mathrm{kind}) \vdash_{\overline{59}} [\,] \to \mathrm{lower\_bound}\,(\mathrm{kind})$$

$$\frac{\nu \not\lhd \mathsf{Value}\ \_ \mid \_\_ \,..\, \_\_ \qquad x, \mathrm{T} \vdash_{\overline{59}} \Sigma \to e}{x, \mathrm{T} \vdash_{\overline{59}} \{\nu\} :: \Sigma \to e}$$

Borne supérieure du premier intervalle dans une liste de contraintes

$$x, \mathrm{T} \vdash_{\overline{60}} \{\mathsf{Value}\,(v)\} :: \Sigma \to \mathsf{Val}\,(v) \qquad x, \mathrm{T} \vdash_{\overline{60}} \{e_0 b_0 \,..\, b_1 e_1\} :: \Sigma \to e_1$$

$$x, \mathsf{INTEGER} \;\; \_ \vdash_{\overline{60}} [\,] \to \mathsf{MAX}$$

$$\text{"REAL"}, \mathrm{T} \vdash_{\overline{60}} [\,] \to \mathsf{Val}\,(\mathsf{PLUS\text{-}INFINITY})$$

$$x, \mathsf{CharString}\,(\mathrm{kind}) \vdash_{\overline{60}} [\,] \to \mathrm{upper\_bound}\,(\mathrm{kind})$$

$$\frac{\nu \not\lhd \mathsf{Value}\ \_ \mid \_\_ \,..\, \_\_ \qquad x, \mathrm{T} \vdash_{\overline{59}} \Sigma \to e}{x, \mathrm{T} \vdash_{\overline{59}} \{\nu\} :: \Sigma \to e}$$

9.6.3 Intervalles bien formés

$$\vdash_{\overline{42}} \mathsf{MIN}\ \_ \,..\, \_\ \mathsf{MAX} \qquad \vdash_{\overline{42}} \mathsf{MIN}\ \_ \,..\, \_\ (\mathsf{Val}\,(\mathsf{Int}\ \_))$$

$$\vdash_{\overline{42}} (\mathsf{Val}\,(\mathsf{Int}\ \_))\ \_ \,..\, \_\ \mathsf{MAX} \qquad \frac{n \leqslant m}{\vdash_{\overline{42}} (\mathsf{Val}\,(\mathsf{Int}\, n)) \leqslant \,..\, \leqslant (\mathsf{Val}\,(\mathsf{Int}\, m))}$$

$$\frac{n < m}{\vdash_{42} (\mathsf{Val}\,(\mathsf{Int}\,n)) \leqslant .. < (\mathsf{Val}\,(\mathsf{Int}\,m))}$$

$$\frac{n < m}{\vdash_{42} (\mathsf{Val}\,(\mathsf{Int}\,n)) < .. \leqslant (\mathsf{Val}\,(\mathsf{Int}\,m))}$$

$$\frac{n + 1 < m}{\vdash_{42} (\mathsf{Val}\,(\mathsf{Int}\,n)) < .. < (\mathsf{Val}\,(\mathsf{Int}\,m))}$$

$$\frac{\begin{array}{c} f_0 \lhd [(\mathsf{Some}\ \text{"mantissa"}, \mathsf{Int}\,m_0); (\mathsf{Some}\ \text{"base"}, \mathsf{Int}\,b_0); \\ (\mathsf{Some}\ \text{"exponent"}, \mathsf{Int}\,e_0)] \\ f_1 \lhd [(\mathsf{Some}\ \text{"mantissa"}, \mathsf{Int}\,m_1); (\mathsf{Some}\ \text{"base"}, \mathsf{Int}\,b_1); \\ (\mathsf{Some}\ \text{"exponent"}, \mathsf{Int}\,e_1)] \\ m_0 \cdot b_0^{e_0} \leqslant m_1 \cdot b_1^{e_1} \end{array}}{\vdash_{42} (\mathsf{Val}\{f_0\}) \leqslant .. \leqslant (\mathsf{Val}\{f_1\})}$$

$$\frac{\begin{array}{c} f_0 \lhd [(\mathsf{Some}\ \text{"mantissa"}, \mathsf{Int}\,m_0); (\mathsf{Some}\ \text{"base"}, \mathsf{Int}\,b_0); \\ (\mathsf{Some}\ \text{"exponent"}, \mathsf{Int}\,e_0)] \\ f_1 \lhd [(\mathsf{Some}\ \text{"mantissa"}, \mathsf{Int}\,m_1); (\mathsf{Some}\ \text{"base"}, \mathsf{Int}\,b_1); \\ (\mathsf{Some}\ \text{"exponent"}, \mathsf{Int}\,e_1)] \\ m_0 \cdot b_0^{e_0} < m_1 \cdot b_1^{e_1} \end{array}}{\vdash_{42} (\mathsf{Val}\{f_0\})\ \_\ ..\ \_\ (\mathsf{Val}\{f_1\})}$$

$$\frac{\begin{array}{c} f_1 \lhd [(\mathsf{Some}\ \text{"mantissa"}, \mathsf{Int}\,m_1); (\mathsf{Some}\ \text{"base"}, \mathsf{Int}\ \_); \\ (\mathsf{Some}\ \text{"exponent"}, \mathsf{Int}\ \_)] \\ 0 < m_1 \end{array}}{\vdash_{42} (\mathsf{Val}\ 0.0)\ \_\ ..\ \_\ (\mathsf{Val}\{f_1\})}$$

$$\frac{\begin{array}{c} f_0 \lhd [(\mathsf{Some}\ \text{"mantissa"}, \mathsf{Int}\,m_0); (\mathsf{Some}\ \text{"base"}, \mathsf{Int}\ \_); \\ (\mathsf{Some}\ \text{"exponent"}, \mathsf{Int}\ \_)] \\ m_0 < 0 \end{array}}{\vdash_{42} (\mathsf{Val}\{f_0\})\ \_\ ..\ \_\ (\mathsf{Val}\ 0.0)}$$

$$\frac{\begin{array}{c} f_1 \lhd [(\mathsf{Some}\ \text{"mantissa"}, \mathsf{Int}\ \_); (\mathsf{Some}\ \text{"base"}, \mathsf{Int}\ \_); \\ (\mathsf{Some}\ \text{"exponent"}, \mathsf{Int}\ \_)] \end{array}}{\vdash_{42} (\mathsf{Val}\ \mathsf{MINUS\text{-}INFINITY})\ \_\ ..\ \_\ (\mathsf{Val}\{f_1\})}$$

$$\frac{\begin{array}{c} f_0 \lhd [(\mathsf{Some}\ \text{"mantissa"}, \mathsf{Int}\ \_); (\mathsf{Some}\ \text{"base"}, \mathsf{Int}\ \_); \\ (\mathsf{Some}\ \text{"exponent"}, \mathsf{Int}\ \_)] \end{array}}{\vdash_{42} (\mathsf{Val}\{f_0\})\ \_\ ..\ \_\ (\mathsf{Val}\ \mathsf{PLUS\text{-}INFINITY})}$$

$$\vdash_{\overline{42}} (\mathsf{Val}\ \mathsf{MINUS\text{-}INFINITY})\ \_\ ..\ \_\ (\mathsf{Val}\ \mathsf{PLUS\text{-}INFINITY})$$

$$\frac{\begin{array}{c}\mathrm{String.length}\, s_0 = 1 \qquad \mathrm{String.length}\, s_1 = 1 \\ \vdash_{\overline{42}} \mathsf{Val}\,(\mathsf{Int}\,(\mathrm{Char.code}\,(s_0.[0]))) \leqslant .. \leqslant \mathsf{Val}\,(\mathsf{Int}\,(\mathrm{Char.code}\,(s_1.[0])))\end{array}}{\vdash_{\overline{42}} \mathsf{Val}\,(\mathsf{CharString}\,(s_0)) \leqslant .. \leqslant \mathsf{Val}\,(\mathsf{CharString}\,(s_1))}$$

$$\vdash_{\overline{42}} e \leqslant .. \leqslant e$$

9.7 Réduction des contraintes

9.7.1 Réduction des intersections de contraintes SIZE

$$\frac{\begin{array}{c}\Gamma(x) \lhd (\alpha, \tau, \{(\mathrm{T}_k, \{\nu_k\})\}_{1 \leqslant k \leqslant q}) \qquad \forall k \in [1..q].x, \mathrm{T}_k \vdash_{\overline{91}} \nu_k \to \overline{\nu}_k \\ \vdash_{\overline{90}} \langle \mathcal{A} \rangle \Gamma \to \langle \overline{\mathcal{A}} \rangle \overline{\Gamma} \qquad \overline{\gamma} \triangleq x \mapsto (\alpha, \tau, \{(\mathrm{T}_k, \{\overline{\nu}_k\})\}_{1 \leqslant k \leqslant q})\end{array}}{\vdash_{\overline{90}} \langle \{x\} \uplus \mathcal{A} \rangle \Gamma \to \langle \overline{\mathcal{A}} \rangle \overline{\Gamma} \oplus \overline{\gamma}}$$

$$\vdash_{\overline{90}} \langle \varnothing \rangle \Gamma \to \langle \varnothing \rangle \Gamma \qquad \frac{\begin{array}{c}\mathsf{None} \vdash_{\overline{74}} \Sigma \to \Sigma_0 \\ \vdash_{\overline{72}} \Sigma_0 \to \nu_0 \qquad x, \mathrm{T} \vdash_{\overline{54}} \nu_0 \mid [\,] \to \nu_1 \qquad \vdash_{\overline{92}} \nu_1 \to \overline{\nu}\end{array}}{x, \mathrm{T} \vdash_{\overline{91}} \mathsf{Inter}\,(\Sigma) \to \overline{\nu}}$$

$$\frac{\nu \not\lhd \mathsf{Inter}\ \_}{x, \mathrm{T} \vdash_{\overline{91}} \nu \to \nu}$$

9.7.2 Normalisation des intersections de contraintes SIZE

$$\frac{\mathsf{Some}\,(\nu) \vdash_{\overline{74}} \Sigma \to \overline{\Sigma}}{\mathsf{None} \vdash_{\overline{74}} \{\mathsf{SIZE}\,\{\nu\}\} :: \Sigma \to \overline{\Sigma}}$$

$$\frac{\mathsf{None} \vdash_{\overline{74}} \mathrm{merge}\,(\nu_0, \nu_1) \to \overline{\Sigma}_0 \qquad \mathsf{Some}\,(\mathsf{Inter}\,\overline{\Sigma}_0) \vdash_{\overline{74}} \Sigma \to \overline{\Sigma}_1}{\mathsf{Some}\,(\nu_1) \vdash_{\overline{74}} \{\mathsf{SIZE}\,\{\nu_0\}\} :: \Sigma \to \overline{\Sigma}_1}$$

$$\frac{\nu \not\lhd \mathsf{SIZE}\ \_ \qquad s \vdash_{\overline{74}} \Sigma \to \overline{\Sigma}}{s \vdash_{\overline{74}} \{\nu\} :: \Sigma \to \{\nu\} :: \overline{\Sigma}} \qquad \mathsf{Some}\,(\nu) \vdash_{\overline{74}} [\,] \to [\{\mathsf{SIZE}\,\{\nu\}\}]$$

$$\mathsf{None} \vdash_{\overline{74}} [\,] \to [\,]$$

9.7.3 Élimination des contraintes FROM si SIZE(0)

$$\frac{\vdash_{\overline{76}} \Sigma \to \overline{\Sigma} \qquad \vdash_{\overline{72}} \overline{\Sigma} \to \overline{\nu}}{\vdash_{\overline{92}} \mathsf{Inter}\,(\Sigma) \to \overline{\nu}} \qquad \frac{\nu \not\lhd \mathsf{Inter}\ \_}{\vdash_{\overline{92}} \nu \to \nu}$$

$$\frac{\{\mathsf{SIZE}\,\{\mathsf{Value}\,(\mathsf{Int}\,0)\}\} \in \Sigma \qquad \vdash_{\overline{78}} \Sigma \to \overline{\Sigma}}{\vdash_{\overline{76}} \Sigma \to \overline{\Sigma}}$$

$$\frac{\begin{array}{c}\{\mathsf{SIZE}\,\{e_0\,b_0..b_1\,e_1\}\} \in \Sigma \\ \nu \triangleq \mathsf{Val}\,(\mathsf{Int}\,0) \leqslant .. \leqslant \mathsf{Val}\,(\mathsf{Int}\,0) \qquad \vdash_{\overline{69}} \nu \equiv e_0 b_0..b_1 e_1 \qquad \vdash_{\overline{78}} \Sigma \to \overline{\Sigma}\end{array}}{\vdash_{\overline{76}} \Sigma \to \overline{\Sigma}}$$

$$\frac{\begin{array}{c}\forall \sigma \in \Sigma.\sigma \not\lhd \{\mathsf{SIZE}\,\{\mathsf{Value}\,(\mathsf{Int}\,0)\}\} \\ \forall \sigma \in \Sigma.(\sigma \lhd \{\mathsf{SIZE}\,\{e_0\,b_0..b_1\,e_1\}\} \Rightarrow \neg(\vdash_{\overline{69}} \nu \equiv e_0 b_0..b_1 e_1))\end{array}}{\vdash_{\overline{76}} \Sigma \to \Sigma}$$

$$\frac{\vdash_{\overline{78}} \Sigma \to \overline{\Sigma}}{\vdash_{\overline{78}} \{\mathsf{FROM}\,\sigma\} :: \Sigma \to \overline{\Sigma}} \qquad \frac{\nu \not\lhd \mathsf{FROM}\ \_ \qquad \vdash_{\overline{78}} \Sigma \to \overline{\Sigma}}{\vdash_{\overline{78}} \{\nu\} :: \Sigma \to \{\nu\} :: \overline{\Sigma}} \qquad \vdash_{\overline{78}} [\,] \to [\,]$$

9.7.4 Réduction des contraintes

$$\frac{\begin{array}{c}\Gamma(x) \lhd (\alpha, \tau, \{(\mathrm{T}_k, \{\nu_k\})\}_{1 \leqslant k \leqslant q}) \qquad \forall k \in [1..q].x,\ \mathrm{T}_k \vdash_{\overline{54}} \nu_k \mid [\,] \to \overline{\nu}_k \\ \vdash_{\overline{94}} \langle \mathcal{A} \rangle \Gamma \to \langle \overline{\mathcal{A}} \rangle \overline{\Gamma} \qquad \overline{\gamma} \triangleq x \mapsto (\alpha, \tau, \{(\mathrm{T}_k, \{\overline{\nu}_k\})\}_{1 \leqslant k \leqslant q})\end{array}}{\vdash_{\overline{94}} \langle \{x\} \uplus \mathcal{A} \rangle \Gamma \to \langle \overline{\mathcal{A}} \rangle \overline{\Gamma} \oplus \overline{\gamma}}$$

$$\vdash_{\overline{94}} \langle \varnothing \rangle \Gamma \to \langle \varnothing \rangle \Gamma$$

$$\frac{x, \mathrm{T} \vdash_{\overline{54}} \nu \mid [\,] \to \overline{\nu}_0 \qquad x, \mathrm{T} \vdash_{\overline{54}} \mathsf{Inter}\,(\Sigma) \mid [\{\overline{\nu}_0\}] \to \overline{\nu}_1}{x, \mathrm{T} \vdash_{\overline{54}} \mathsf{Inter}\,(\{\nu\} :: \Sigma) \mid [\,] \to \overline{\nu}_1}$$

$$\frac{\overline{\Sigma}_0 \not\lhd [\,] \qquad x, \mathrm{T} \vdash_{\overline{54}} \nu \mid \overline{\Sigma}_0 \to \mathsf{Inter}\,(\overline{\Sigma}_1) \qquad x, \mathrm{T} \vdash_{\overline{54}} \mathsf{Inter}\,(\Sigma) \mid \overline{\Sigma}_1 \to \overline{\nu}_1}{x, \mathrm{T} \vdash_{\overline{54}} \mathsf{Inter}\,(\{\nu\} :: \Sigma) \mid \overline{\Sigma}_0 \to \overline{\nu}_1}$$

$$x, \mathrm{T} \vdash_{\overline{54}} \mathsf{Inter}\,[\,] \mid \overline{\Sigma} \to \mathsf{Inter}\,(\overline{\Sigma})$$

$$\frac{\text{"REAL"}, \mathrm{T} \vdash_{\overline{73}} \mathsf{Value}\,(\overline{v}) \mid \overline{\Sigma} \to \overline{\nu} \qquad \overline{v} \triangleq \mathrm{Real.normalize}\,(v)}{\text{"REAL"}, \mathrm{T} \vdash_{\overline{54}} \mathsf{Value}\,(v) \mid \overline{\Sigma} \to \overline{\nu}}$$

$$\frac{x \neq \text{"REAL"} \qquad x, \mathrm{T} \vdash_{73} \mathsf{Value}\,(v) \mid \overline{\Sigma} \to \overline{\nu}}{x, \mathrm{T} \vdash_{54} \mathsf{Value}\,(v) \mid \overline{\Sigma} \to \overline{\nu}}$$

$$\frac{\mathrm{T} \lhd \mathsf{INTEGER} \ \_ \mid \mathsf{CharString}\ \_ \qquad x, \mathrm{T} \vdash_{44} \nu \mid \overline{\Sigma} \to \overline{\nu}}{x, \mathrm{T} \vdash_{54} \_\_..\_\_ \ \mathbf{as}\ \nu \mid \overline{\Sigma} \to \overline{\nu}}$$

$$\frac{x = \text{"REAL"} \qquad \vdash_{50} \nu \to \nu_0 \qquad x, \mathrm{T} \vdash_{44} \nu_0 \mid \overline{\Sigma} \to \overline{\nu}}{x, \mathrm{T} \vdash_{54} \_\_..\_\_ \ \mathbf{as}\ \nu \mid \overline{\Sigma} \to \overline{\nu}}$$

$$\frac{\begin{array}{c}\mathrm{T} \lhd \mathsf{BIT\ STRING}\ \ \_ \mid \mathsf{OCTET\ STRING} \\ \mid (\mathsf{SET\ OF} \mid \mathsf{SEQUENCE\ OF})\_\_\_ \mid \mathsf{CharString}\ \_ \\ \sigma_0 \triangleq \{\mathsf{Inter}\,[\mathsf{Val}\,(\mathsf{Int}\,0) \leqslant .. \leqslant \mathsf{MAX}; \sigma']\} \\ \text{"INTEGER"}, \mathsf{INTEGER}\ [\,] \vdash_{55} \sigma_0 \mid [\,] \to \overline{\sigma}_0 \\ x, \mathrm{T} \vdash_{73} \mathsf{SIZE}\,(\overline{\sigma}_0) \mid \overline{\Sigma} \to \overline{\nu}\end{array}}{x, \mathrm{T} \vdash_{54} \mathsf{SIZE}\,(\sigma') \mid \overline{\Sigma} \to \overline{\nu}}$$

$$\frac{\begin{array}{c}\mathrm{T} \lhd \mathsf{CharString}\ \_ \\ x, \mathrm{T} \vdash_{55} \sigma' \mid [\,] \to \overline{\sigma}' \qquad x, \mathrm{T} \vdash_{73} \mathsf{FROM}\,(\overline{\sigma}') \mid \overline{\Sigma} \to \overline{\nu}\end{array}}{x, \mathrm{T} \vdash_{54} \mathsf{FROM}\,(\sigma') \mid \overline{\Sigma} \to \overline{\nu}}$$

$$\frac{x, \mathrm{T} \vdash_{54} \nu \mid [\,] \to \overline{\nu} \qquad x, \mathrm{T} \vdash_{55} \sigma \to \overline{\sigma}}{x, \mathrm{T} \vdash_{55} \{\nu\} \uplus \sigma \to \{\overline{\nu}\} \cup \overline{\sigma}} \qquad x, \mathrm{T} \vdash_{55} \{\} \to \{\}$$

$$\frac{x, \mathrm{T} \vdash_{45} \nu \mid \overline{\Sigma} \to \nu_0 \qquad \vdash_{42} \nu_0 \qquad x, \mathrm{T} \vdash_{73} \nu_0 \mid \overline{\Sigma} \to \overline{\nu}}{x, \mathrm{T} \vdash_{44} \nu \mid \overline{\Sigma} \to \overline{\nu}}$$

9.7.5 Intersection de contraintes

$$\frac{x, \mathrm{T} \vdash_{70} \nu \mid \Sigma \to \overline{\Sigma} \qquad \vdash_{72} \overline{\Sigma} \to \overline{\nu}}{x, \mathrm{T} \vdash_{73} \nu \mid \Sigma \to \overline{\nu}}$$

$$\frac{\begin{array}{c}\mathrm{T} \lhd \mathsf{INTEGER}\ \ \_ \mid \mathsf{CharString}\ \_ \\ \vee\ x = \text{"REAL"} \qquad \nu_0 \lhd \_\_..\_\_ \mid \mathsf{Value}\ \_ \\ \nu_1 \lhd \_\_..\_\_ \mid \mathsf{Value}\ \_ \qquad \vdash_{71} \nu_0 \cap \nu_1 \to \nu_2\end{array}}{x, \mathrm{T} \vdash_{70} \nu_0 \mid [\{\nu_1\}] \sqcup \overline{\Sigma} \to [\{\nu_2\}] \sqcup \overline{\Sigma}}$$

$$\frac{\{\mathsf{Value}\,(v)\} \in \overline{\Sigma}}{x, \mathrm{T} \vdash_{70} \mathsf{Value}\,(v) \mid \overline{\Sigma} \to \overline{\Sigma}}$$

$$\frac{\vdash_{\overline{72}} \mathrm{merge}\,(\nu_0,\nu_1) \to \nu_2 \qquad x, \mathrm{T} \vdash_{\overline{54}} \mathsf{SIZE}\,\{\nu_2\} \mid [\,] \to \overline{\nu}}{x, \mathrm{T} \vdash_{\overline{70}} \mathsf{SIZE}\,\{\nu_0\} \mid [\{\mathsf{SIZE}\,\{\nu_1\}\}] \sqcup \overline{\Sigma} \to [\{\overline{\nu}\}] \sqcup \overline{\Sigma}}$$

$$\frac{\vdash_{\overline{72}} \mathrm{merge}\,(\nu_0,\nu_1) \to \nu_2 \qquad x, \mathrm{T} \vdash_{\overline{54}} \mathsf{FROM}\,\{\nu_2\} \mid [\,] \to \overline{\nu}}{x, \mathrm{T} \vdash_{\overline{70}} \mathsf{FROM}\,\{\nu_0\} \mid [\{\mathsf{FROM}\,\{\nu_1\}\}] \sqcup \overline{\Sigma} \to [\{\overline{\nu}\}] \sqcup \overline{\Sigma}}$$

$$\frac{\begin{array}{c}(\nu_0,\nu_1) \not\triangleleft ((\_\_..\_\_ \mid \mathsf{Value}\ \_), (\_\_..\_\_ \mid \mathsf{Value}\ \_)) \\ \mid (\mathsf{SIZE}\ \_, \mathsf{SIZE}\ \_) \mid (\mathsf{FROM}\ \_, \mathsf{FROM}\ \_) \\ x, \mathrm{T} \vdash_{\overline{70}} \nu_0 \mid \Sigma \to \overline{\Sigma}_0\end{array}}{x, \mathrm{T} \vdash_{\overline{70}} \nu_0 \mid [\{\nu_1\}] \sqcup \overline{\Sigma} \to [\{\nu_1\}] \sqcup \overline{\Sigma}_0}$$

$$x, \mathrm{T} \vdash_{\overline{70}} \nu_0 \mid [\,] \to [\{\nu_0\}] \qquad \vdash_{\overline{72}} [\{\nu\}] \to \nu \qquad \frac{\Sigma \not\triangleleft [\_]}{\vdash_{\overline{72}} \Sigma \to \mathsf{Inter}\,(\Sigma)}$$

9.8 Sous-types bien formés

$$\frac{\begin{array}{c}\vdash_{\overline{89}} \langle\mathcal{A}\rangle\Gamma \to \langle\mathcal{A}_0\rangle\Gamma_0 \qquad \vdash_{\overline{62}} \langle\mathcal{A}_0\rangle\Gamma_0 \to \langle\mathcal{A}_1\rangle\Gamma_1 \\ \langle\mathcal{B}\rangle\Delta \vdash_{\overline{43}} \langle\mathcal{A}_1\rangle\Gamma_1 \to \langle\mathcal{A}_2\rangle\Gamma_2 \qquad \vdash_{\overline{20}} \langle\mathcal{A}_2\rangle\Gamma_2 \to \langle\mathcal{A}_3\rangle\Gamma_3 \\ \Vdash \langle\mathcal{A}_3\rangle\Gamma_3 \qquad \vdash_{\overline{90}} \langle\mathcal{A}_3\rangle\Gamma_3 \to \langle\mathcal{A}_4\rangle\Gamma_4 \qquad \vdash_{\overline{94}} \langle\mathcal{A}_4\rangle\Gamma_4 \to \_\end{array}}{\langle\mathcal{B}\rangle\Delta \vdash_{\overline{93}} \langle\mathcal{A}\rangle\Gamma}$$

Chapitre 10

Analyse lexico-syntaxique

Maître de philosophie. — On les peut mettre [les paroles] premièrement comme vous avez dit : *Belle Marquise, vos beaux yeux me font mourir d'amour*, ou bien : *D'amour mourir me font, belle Marquise, vos beaux yeux.* Ou bien : *Vos yeux beaux d'amour me font, belle Marquise, mourir.* Ou bien : *Mourir vos beaux yeux, belle Marquise, d'amour me font.* Ou bien : *Me font vos yeux beaux mourir, belle Marquise, d'amour.*

Monsieur Jourdain. — Mais de toutes ces façons-là laquelle est la meilleure ?

Maître de philosophie. — Celle que vous avez dite : *Belle Marquise, vos beaux yeux me font mourir d'amour.*

Molière. *Le bourgeois gentilhomme,* Acte II, Scène IV.

Jusqu'à présent, les compilateurs ASN.1 sont déficients quant à leur analyse syntaxique. En effet, ils ne parviennent pas à reconnaître toute la grammaire du langage, et par conséquent, imposent des restrictions par rapport à celle-ci, ou bien ont un comportement inattendu et celé. Nous montrons que l'on peut obtenir une grammaire ASN.1 dans sa version de 1990 ayant la propriété LL(1) (section 10.3 page 246), grâce à une suite de transformations à partir de la forme normalisée, sans faire aucun compromis avec celle-ci. Une grammaire peut être ici considérée comme étant un ensemble d'équations rationnelles ayant pour (plus petite) solution un langage non contextuel. Chaque transformation est donc décrite en fonction d'opérations rationnelles (union, produit et étoile) garantissant l'invariance du langage engendré par la grammaire résultante.

Les macros ASN.1 sont traitées mais pas dans toute leur généralité, étant donné que cela est irréalisable. Les restrictions apportées permettent de réaliser en OCaml un analyseur syntaxique complet fonctionnant en *une passe*.

Tout d'abord nous présentons une grammaire LL(1) pour le lexique d'ASN.1, obtenue à partir des indications en langue naturelle du document ISO. Ensuite vient l'étude de la syntaxe d'ASN.1. La contrainte qui guidait le choix des transformations était de rendre analysable de façon descendante la grammaire finale, ce qui impliquait qu'elle devait être non ambiguë. Or, la question de l'ambiguïté d'une grammaire étant un problème indécidable, il a fallu faire des choix explicites *ad hoc* pour résoudre les difficultés rencontrées. De plus, nous ne savions même pas *a priori*, en supposant éliminées ces difficultés, s'il existait une grammaire LL(1) pour ASN.1. Ces deux raisons rendirent nécessaire un travail « à la main ». D'autre part, étant données deux grammaires, le problème de l'égalité des langages engendrés par celles-ci est aussi indécidable (A. Aho et J. Ullman, 1972). Par conséquent, la preuve de cette égalité se fera par la présentation *in extenso*, étape par étape, de chaque transformation syntaxique (cf. annexe pour tous les détails). Pour faciliter la compréhension, la grammaire initiale sera découpée en sections dont nous suivrons l'évolution. La preuve de la propriété LL(1) est donnée ensuite. Les errata de la norme ont été pris en compte dès les étapes initiales des transformations, leur validité étant rétroactive. Ils sont signalés dès leur première apparition.

10.1 Présentation du formalisme syntaxique

10.1.1 Les grammaires algébriques

La notation pour les grammaires adoptée dans ce document est *grosso modo* la BNF (*Backus-Naur Form*), plus quelques ajouts (notamment d'opérateurs rationnels).

Lexique

- Les identificateurs de terminaux apparaissent en minuscules.
- Les mots-clefs sont en majuscules.
- Les identificateurs de non-terminaux sont la concaténation d'identificateurs en minuscules, débutant chacun par une majuscule.
- Les extraits de syntaxe concrète (terminaux) sont entre guillemets — si ces derniers font partie de la syntaxe concrète, alors ils sont précédés d'une contre-oblique (anglais, *backslash*).

Grammaires

Nous appellerons *règle de production* (ou, en abrégé, *règle*), le couple formé par un non-terminal et tous les mots qui peuvent en être dérivés en une étape. Nous nommerons *membre droit d'une règle de production* (ou ,en abrégé, *membre droit*) un des mots (constitué de terminaux ou non) que l'on peut dériver en une étape. Par exemple, $X_0X_1 \dots X_n$ est un membre droit de la règle de production X dans

$$X \to X_0X_1 \dots X_n \mid \dots$$

Nous baptiserons *production* l'ensemble des membres droits d'une règle de production.

Structure La grammaire est découpée en sections qui sont divisées en sous-sections séparées par une ligne. Le but de ce découpage est de regrouper des règles qui ont une sémantique voisine ou qui contribuent à une même sémantique. Une sous-section A séparée par une ligne double d'une sous-section B signifie que B est une transformée de A. L'ordre des règles au sein d'une même sous-section est significatif : c'est un parcours en largeur à partir d'une règle d'entrée, en considérant les productions dans l'ordre d'écriture. L'ordre entre les règles d'entrée n'est pas significatif.

Les règles d'entrée sont des règles qui sont appelées en dehors de la sous-section où elles sont définies. Si les appels sont restreints à la section courante, la règle est dite *locale* (à la section), s'ils sont limités au dehors de la section définissante, elle est dite *globale*, et s'il existe des appels à la fois dans la section courante et dans les autres sections, elle est qualifiée de *mixte*. Dans le premier cas, l'identificateur du non-terminal définissant apparaîtra en italique, dans le deuxième en souligné, et dans le dernier en italique souligné. L'axiome sera en gras. Il peut y avoir plusieurs règles d'entrée dans une sous-section.

Commentaires Dans la présentation des grammaires intermédiaires, les commentaires seront dans des boîtes, après les règles. Il est impératif de les lire dans l'ordre d'apparition dans la sous-section, car ils peuvent décrire des transformations composées (d'où le séquencement). Pour la même raison, il faut lire les sections dans l'ordre d'apparition. D'autre part, les règles créées par une transformation seront données entre parenthèses.

Conventions En transformant une règle se créent de nouvelles règles pour lesquelles il est difficile de trouver un identificateur pertinent suggérant sa sémantique. Dans ce cas, chacune de ces règles aura

pour identificateur la concaténation de celui de la règle appelante (parfois abrégé), plus un préfixe et/ou un suffixe, souvent numéroté. Dans les exemples contenus dans ce document, nous adopterons en outre les conventions lexicographiques supplémentaires suivantes :

— La production vide est notée ε.
— Une lettre latine minuscule représente toujours un terminal.
— Une lettre latine majuscule représente toujours un non-terminal.
— Une lettre grecque minuscule représente toujours une concaténation quelconque de terminaux et de non-terminaux.
— Nous noterons $\mathrm{A} \Longrightarrow \alpha$ la relation « A produit α ».

Opérateurs rationnels

Les opérateurs rationnels suivants ont été rajouté à la notation BNF, pour gagner en compacité et lisibilité (il ne s'agit pas d'un gain d'expressivité) : α^*, α^+, $[\alpha]$, { A a ... }*, { A a ... }$^+$. Chaque occurrence de ces expressions peut être remplacée par un non-terminal dont la règle définissante est spécifique. Voici le tableau donnant leur définition .

Notation	Définition	Contrainte de validité
$\mathrm{X} \to \alpha^*$	$\mathrm{X} \to \alpha\ \mathrm{X} \mid \varepsilon$	$\neg(\alpha \overset{*}{\Longrightarrow} \varepsilon)$
$\mathrm{X} \to \alpha^+$	$\mathrm{X} \to \alpha \mathrm{A}^*$	$\neg(\alpha \overset{*}{\Longrightarrow} \varepsilon)$
$\mathrm{X} \to [\alpha]$	$\mathrm{X} \to \alpha \mid \varepsilon$	$\neg(\alpha \overset{*}{\Longrightarrow} \varepsilon)$
$\mathrm{X} \to \{\mathrm{A\ a} \ldots\}^*$	$\mathrm{X} \to \varepsilon \mid \mathrm{A\ (a\ A)}^*$	$\neg(\mathrm{A} \overset{*}{\Longrightarrow} \varepsilon)$
$\mathrm{X} \to \{\mathrm{A\ a} \ldots\}^+$	$\mathrm{X} \to \mathrm{A\ (a\ A)}^*$	$\neg(\mathrm{A} \overset{*}{\Longrightarrow} \varepsilon)$

Les définitions choisie sont LL(1).

10.1.2 Les transformations de grammaires

Ce sont des applications de propriétés de base des expressions rationnelles, exprimées avec des notations un peu différentes dans le cadre des grammaires algébriques.

La factorisation forme des préfixes ou des suffixes (ou les deux à la fois) d'un ensemble de productions d'une même règle.

— **Préfixe**

$$\mathrm{X} \to \beta\gamma_1 \mid \beta\gamma_2 \mid \ldots \mid \beta\gamma_n \mid \mathrm{B} \quad \textit{devient} \quad \begin{array}{l} \mathrm{X} \to \beta\mathrm{Y} \mid \mathrm{B} \\ \mathrm{Y} \to \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_n \end{array}$$

— **Suffixe**

$$\mathrm{X} \to \alpha_1\beta \mid \alpha_2\beta \mid \ldots \mid \alpha_n\beta \mid \mathrm{B} \quad \textit{devient} \quad \begin{array}{l} \mathrm{X} \to \mathrm{Y}\beta \mid \mathrm{B} \\ \mathrm{Y} \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n \end{array}$$

— **Bifixe**

$$\mathrm{X} \to \alpha\beta_1\gamma \mid \alpha\beta_2\gamma \mid \ldots \mid \alpha\beta_n\gamma \mid \mathrm{B} \quad \textit{devient} \quad \begin{array}{l} \mathrm{X} \to \alpha\mathrm{Y}\gamma \mid \mathrm{B} \\ \mathrm{Y} \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n \end{array}$$

Nous pouvons appliquer ces factorisations si $n = 1$, ou sur une partie seulement des productions factorisables.

La réduction regroupe des productions d'une même règle pour former un nouveau non-terminal :

$$\mathrm{X} \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n \quad \textit{devient} \quad \begin{array}{lcl} \mathrm{X} & \to & \mathrm{Y} \mid \alpha_{i+1} \mid \alpha_{i+2} \mid \ldots \mid \alpha_n \\ \mathrm{Y} & \to & \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_i \end{array}$$

L'élimination Toute règle devenue inutile (typiquement à la suite d'une expansion globale) peut être supprimée. Une règle d'entrée globale ou mixte, devenue inutile au sein d'une section, n'y figurera plus par la suite — ce qui ne signifie pas son élimination de la grammaire ! Nous pouvons aussi éliminer les productions redondantes au sein d'une même règle pour n'en conserver qu'une.

Remarque 1 La transformation suivante n'est *pas* une élimination :

$\mathrm{X} \to \mathrm{X} \mid \mathrm{A}$ *devient* $\mathrm{X} \to \mathrm{A}$

Cf. le paragraphe consacré au lemme d'Arden (10.1.2 page 237).

Remarque 2 L'élimination suivante est licite :

$\mathrm{X} \to \texttt{lower}_{val} \mid \texttt{lower}_{id}$ *devient* $\mathrm{X} \to \texttt{lower}$

Le renommage Pour des raisons de lisibilité, les identificateurs peuvent être renommés globalement dans toute la grammaire. Nous tâcherons d'éviter cette manipulation en choisissant dès le départ des noms adaptés pour éviter les collisions.

L'expansion peut être totale, partielle, préfixe, suffixe ou globale.

Totale Elle consiste à substituer textuellement tout le membre droit d'une règle à *une* occurrence du non-terminal correspondant. Par exemple :

$$\begin{array}{lcl} \mathrm{A} & \to & \mathrm{a\ B\ C} \\ & \mid & \mathrm{B} \\ \mathrm{B} & \to & \mathrm{b\ A} \\ & \mid & \mathrm{b} \\ \mathrm{C} & \to & \mathrm{c} \end{array} \quad \textit{devient} \quad \begin{array}{lcl} \mathrm{A} & \to & \mathrm{a\ b\ A\ C} \\ & \mid & \mathrm{a\ b\ C} \\ & \mid & \mathrm{B} \\ \mathrm{B} & \to & \mathrm{b\ A} \\ & \mid & \mathrm{b} \\ \mathrm{C} & \to & \mathrm{c} \end{array}$$

Ce sera l'expansion par défaut, en l'absence de qualificatif.

Globale C'est une expansion **totale** d'une règle pour toutes les occurrences possibles dans toute la grammaire (cette règle peut alors

être éliminée). La présentation des transformations se faisant par modules, le lecteur pourrait être dérouté par cette portée globale. Concrètement, une expansion globale équivaudra à une expansion totale appliquée à toute la sous-section, suivie d'une élimination ; sinon une note sera fournie.

Partielle C'est la composition d'une expansion **globale**, d'une factorisation bifixe partielle et d'un renommage. En résumé :

$$\begin{array}{lcl} \text{Z} & \to & \alpha\ \text{A}\ \beta \\ \text{A} & \to & \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_n \end{array}$$

devient

$$\begin{array}{lcl} \text{Z} & \to & \alpha\gamma_1\beta \mid \alpha\gamma_2\beta \mid \ldots \mid \alpha\gamma_{i-1}\beta \mid \alpha\ \text{A}\ \beta \\ \text{A} & \to & \gamma_i \mid \gamma_{i+1} \mid \ldots \mid \gamma_n \end{array}$$

Préfixe C'est la composition d'une expansion **globale** d'une règle factorisable à gauche, d'une factorisation bifixe et d'un renommage. Schématiquement :

$$\begin{array}{lcl} \text{Z} & \to & \alpha\ \text{A}\ \beta \\ \text{A} & \to & \gamma\alpha_1 \mid \gamma\alpha_2 \mid \ldots \mid \gamma\alpha_n \end{array}$$

devient

$$\begin{array}{lcl} \text{Z} & \to & \alpha\gamma\ \text{A}\ \beta \\ \text{A} & \to & \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n \end{array}$$

Suffixe C'est la composition d'une expansion **globale** d'une règle factorisable à droite, d'une factorisation bifixe et d'un renommage. C'est-à-dire :

$$\begin{array}{lcl} \text{Z} & \to & \alpha\ \text{A}\ \beta \\ \text{A} & \to & \alpha_1\gamma \mid \alpha_2\gamma \mid \ldots \mid \alpha_n\gamma \end{array}$$

devient

$$\begin{array}{lcl} \text{Z} & \to & \alpha\ \text{A}\ \gamma\beta \\ \text{A} & \to & \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n \end{array}$$

L'option C'est le cas particulier de l'expansion **partielle** d'une production vide, suivie d'une « mise entre crochets ». Par exemple :

$$\begin{array}{lcl} \text{A} & \to & \text{a B C} \\ & \mid & \text{B} \\ \text{B} & \to & \text{b A} \\ & \mid & \varepsilon \\ \text{C} & \to & \text{c} \end{array} \quad \textit{devient} \quad \begin{array}{lcl} \text{A} & \to & \text{a [B] C} \\ & \mid & \text{[B]} \\ \text{B} & \to & \text{b A} \\ \text{C} & \to & \text{c} \end{array}$$

De même, nous assimilerons à une option (par raccourci) la transformation :

$$\begin{array}{lcl} \text{X} & \to & \text{A} \\ & \mid & \text{b} \\ \text{A} & \to & \{\ \text{B “,” } \ldots\}^* \\ & \mid & \text{[a B c]} \\ & \mid & \text{a} \end{array} \quad \textit{devient} \quad \begin{array}{lcl} \text{X} & \to & \text{[A]} \\ & \mid & \text{b} \\ \text{A} & \to & \{\ \text{B “,” } \ldots\}^+ \\ & \mid & \text{a B c} \\ & \mid & \text{a} \end{array}$$

Dans le but de faciliter la preuve de la propriété LL(1) de la grammaire finale, nous appliquerons l'option chaque fois que cela sera possible. Ainsi, la forme résultante ne contiendra plus de productions vides *explicites* (attention : nous ne faisons *pas* disparaître ces productions !). En d'autres termes, nous n'autorisons aucune règle à produire explicitement ε.

Notons que les réciproques des factorisations, des réductions, des expansions et des options sont aussi des transformations licites.

Le lemme d'Arden

Supposons que nous ayons une équation rationnelle de langages de la forme $X = \alpha X + \beta$, avec $\varepsilon \notin \alpha$, et où + représente la disjonction (|). Alors le lemme d'Arden affirme que $X = \alpha^*\beta$ est la solution unique de l'équation. Si $\varepsilon \in \alpha$, alors $X = \alpha^*(\beta+\gamma)$ est solution, quelque soit γ (qui peut même être un langage contextuel). Nous avons donc une infinité de solutions, et nous choisirons dans ce cas le plus petit langage solution (le point fixe minimal) : $X = \alpha^*\beta$. En résumé, nous autoriserons toujours la transformation : $X \to \alpha X \mid \beta$ devient $X \to \alpha^*\beta$. Nous avons ainsi le cas particulier $X \to X \mid \beta$ devient $X \to \beta$. Nous nommerons *Arden* toutes les transformations élémentaires qui impliquent l'application du lemme d'Arden :

$X \to \alpha\ X \mid \varepsilon$ *ou* $X \to X\ \alpha \mid \varepsilon$	*devient*	$X \to \alpha^*$
$X \to \alpha\ X \mid \alpha$ *ou* $X \to X\ \alpha \mid \alpha$	*devient*	$X \to \alpha^+$
$X \to A \mid A\ a\ X$	*devient*	$X \to \{A\ a\ \ldots\}^+$
$X \to \varepsilon \mid A \mid A\ a\ X$	*devient*	$X \to \{A\ a\ \ldots\}^*$

Remarque 1 Nous avons : $X \to X\alpha \mid \beta$ devient $X \to \beta\alpha^*$.

Remarque 2 Le lemme d'Arden permet d'écrire des égalités non triviales, comme par exemple : $(a+cb^*d)^* = a^*+a^*c(b+da^*c)^*da^*$.

Ambiguïtés syntaxiques

L'ambiguïté d'une grammaire est toujours un problème épineux pour l'analyse du langage qu'elle engendre. Une grammaire est ambiguë si

et seulement si on peut construire deux dérivations différentes pour un même mot du langage. Une condition suffisante d'ambiguïté est la double récursivité d'une règle. Quand la double récursivité est au sein d'une même production, nous pouvons parfois régler ce problème en appliquant le lemme d'Arden. Un cas typique où apparaît cette double récursivité est celui des grammaires d'expressions arithmétiques, où des opérateurs sont binaires et infixes : E → E "+" E | E "∗" E | "(" E ")" | **id**. Si la double récursivité est à gauche et à droite, sur deux productions, nous pouvons procéder comme suit. Supposons : Z → AZ | ZB | C. Le lemme d'Arden donne ici : Z → AZB* | CB*. Or, il est trivial que : X → αXβ | γ devient $\forall n \geq 0$.X → $\alpha^n\gamma\beta^n$. D'où : $\forall n \geq 0$.Z → $A^n(CB^*)(B^*)^n$, qui se simplifie en Z → A*CB*. En réécrivant $Z \rightarrow A^+CB^*$ | CB*, il vient Z → AZ | CB*. En résumé, nous retiendrons la transformation suivante :
Z → AZ | ZB | C devient Z → AZ | CB*

10.1.3 Lexique d'ASN.1

Une grammaire pour le lexique d'ASN.1

À partir de la norme nous pouvons extraire une grammaire du lexique (non LL(1)). Pour plus de détails voir le code de l'analyseur lexical en annexe.

Lower	→	“a” | “b” | . . . | “z”
Upper	→	“A” | “B” | . . . | “Z”
Letter	→	Lower | Upper
Digit	→	“0” | “1” | . . . | “9”
Alpha	→	Letter | Digit
ExtAlpha	→	Alpha | extrasym
HexaBin	→	“H” | “B”

Lexer	→	Tokens*
Tokens	→	Blank* Start
Blank	→	“␣” | “\t” | “\n”
Start	→	stdsym
	|	Digit$^+$
	|	“-” [AuxMinus]
	|	“.” [AuxDot]
	|	“ :” [AuxColon]
	|	“\"” AuxString
	|	“’” Alpha* “’” HexaBin
	|	Lower AuxRef
	|	Upper AuxRef

AuxMinus	→	“-” [Comment]
AuxDot	→	“.” [“.”]
AuxColon	→	“ :” [Four]
AuxString	→	ExtAlpha* “\"” [“\"” AuxString]
AuxRef	→	Alpha* “-” (Alpha$^+$ “-”)* AuxMinus

Comment	→	“\n” | ’-’ [AuxCom] | ExtAlpha [Comment]
AuxCom	→	“-” | Comment
Four	→	“=” | AuxColon

Ambiguïtés lexicales

D’après le document ISO, plusieurs terminaux sémantiquement différents sont lexicalement indistinguables. Seul le contexte où ils apparaissent permet de les distinguer. Ainsi, un identificateur de type `typereference` est identique à un identificateur de module `modulereference`.

De même, un identificateur de valeur `valuereference` est identique à un identificateur de champ `identifier` dans un type `SEQUENCE`. Dans la grammaire de la syntaxe d'ASN.1, ils seront dénotés respectivement par les identificateurs `upper` et `lower`. Lorsqu'il n'y a pas d'ambiguïté, nous ferons figurer en indice la véritable nature du terminal :

`typereference`, `modulereference`	$\rightarrow$	$\texttt{upper}_{typ}$, $\texttt{upper}_{mod}$, `upper`
`valuereference`, `identifier`	$\rightarrow$	$\texttt{lower}_{val}$, $\texttt{lower}_{id}$, `lower`

Les terminaux `bstring` et `hstring` ont la même sémantique (qui est celle de dénoter un nombre en base binaire ou hexadécimale), et seront donc fusionnés sous le nom `basednum`. De plus, `cstring` qui dénote une chaîne de caractères à été rebaptisé `string`.

10.2 Nouvelle grammaire d'ASN.1

Nous donnons ici la nouvelle grammaire ASN.1 obtenue par les transformations données en annexe (cf. 10.5 page 292). Elle décrit exactement le même langage que la grammaire normalisée par l'ISO.

MODULES

ModuleDefinition	→	ModuleIdentifier DEFINITIONS [TagDefault TAGS] " : :=” BEGIN [ModuleBody] END
ModuleIdentifier	→	`upper`$_{mod}$ [“{” ObjIdComponent$^+$ “}”]
ObjIdComponent	→	`number`
	|	`upper`$_{mod}$ “.” `lower`$_{val}$
	|	`lower` [“(” ClassNumber “)”]
TagDefault	→	EXPLICIT
	|	IMPLICIT
ModuleBody	→	[Exports] [Imports] Assignment$^+$
Exports	→	EXPORTS {Symbol “,” . . . }* “ ;”
Imports	→	IMPORTS SymbolsFromModule* “ ;”
SymbolsFromModule	→	{Symbol “,” . . . }$^+$ FROM ModuleIdentifier
Symbol	→	`upper`$_{typ}$
	|	`lower`$_{val}$
Assignment	→	`upper`$_{typ}$ “ : :=” Type
	|	`lower`$_{val}$ Type “ : :=” Value

TYPES

Type	→	`lower`$_{id}$ "<" Type
	|	`upper` ["." `upper`$_{typ}$] SubtypeSpec*
	|	NULL SubtypeSpec*
	|	AuxType
AuxType	→	"[" [Class] ClassNumber "]" [TagDefault] Type
	|	BuiltInType SubtypeSpec*
	|	SetSeq [TypeSuf]
SetSeq	→	SET
	|	SEQUENCE
TypeSuf	→	SubtypeSpec$^+$
	|	"{" {ElementType "," ...}* "}" SubtypeSpec*
	|	[SIZE SubtypeSpec] OF Type

BuiltInType	→	BOOLEAN
	|	INTEGER ["{" {NamedNumber "," ...}$^+$ "}"]
	|	BIT STRING ["{" {NamedBit "," ...}$^+$ "}"]
	|	OCTET STRING
	|	CHOICE "{" {NamedType "," ...}$^+$ "}"
	|	ANY [DEFINED BY `lower`$_{id}$]
	|	OBJECT IDENTIFIER
	|	ENUMERATED "{" {NamedNumber "," ...}$^+$ "}"
	|	REAL
	|	"NumericString"
	|	"PrintableString"
	|	"TeletexString"
	|	"T61String"
	|	"VideotexString"
	|	"VisibleString"
	|	"ISO646String"
	|	"IA5String"
	|	"GraphicString"
	|	"GeneralString"
	|	EXTERNAL
	|	"UTCTime"
	|	"GeneralizedTime"
	|	"ObjectDescriptor"

NamedType	→	`lower`$_{id}$ [“<”] Type
	|	`upper` [“.” `upper`$_{typ}$] SubtypeSpec*
	|	NULL SubtypeSpec*
	|	AuxType

NamedNumber	→	`lower`$_{id}$ “(” AuxNamedNum “)”
AuxNamedNum	→	[“-”] `number`
	|	[`upper`$_{mod}$ “.”] `lower`$_{val}$

NamedBit	→	`lower`$_{id}$ “(” ClassNumber “)”

ElementType	→	NamedType [ElementTypeSuf]
	|	COMPONENTS OF Type
ElementTypeSuf	→	OPTIONAL
	|	DEFAULT Value

Class	→	UNIVERSAL
	|	APPLICATION
	|	PRIVATE
ClassNumber	→	`number`
	|	[`upper`$_{mod}$ “.”] `lower`$_{val}$

VALEURS

Value	→	AuxVal0
	|	`upper` AuxVal1
	|	`lower` [AuxVal2]
	|	[“-”] `number`
AuxVal0	→	BuiltInValue
	|	AuxType “ :” Value
	|	NULL [SpecVal]
AuxVal1	→	SpecVal
	|	“.” AuxVal11
AuxVal2	→	[“<” Type] “ :” Value
AuxVal11	→	`upper`$_{typ}$ SpecVal
	|	`lower`$_{val}$
SpecVal	→	SubtypeSpec* “ :” Value

BuiltInValue	→	TRUE
	|	FALSE
	|	PLUS-INFINITY
	|	MINUS-INFINITY
	|	`basednum`
	|	`string`
	|	"{" [BetBraces] "}"

BetBraces	→	AuxVal0 [AuxNamed]
	|	"-" `number` [AuxNamed]
	|	`lower` [AuxBet1]
	|	`upper` AuxBet2
	|	`number` [AuxBet3]
AuxBet1	→	"(" ClassNumber ")" ObjIdComponent*
	|	AuxNamed
	|	AuxVal2 [AuxNamed]
	|	"-" `number` [AuxNamed]
	|	AuxVal0 [AuxNamed]
	|	`lower` [AuxBet11]
	|	`number` [AuxBet3]
	|	`upper` AuxBet2
AuxBet2	→	SpecVal [AuxNamed]
	|	"." AuxBet21
AuxBet3	→	ObjIdComponent$^+$
	|	AuxNamed
AuxBet11	→	"(" ClassNumber ")" ObjIdComponent*
	|	ObjIdComponent$^+$
	|	AuxVal2 [AuxNamed]
	|	AuxNamed
AuxBet21	→	$\texttt{upper}_{typ}$ SpecVal [AuxNamed]
	|	$\texttt{lower}_{val}$ [AuxBet3]
AuxNamed	→	"," {NamedValue "," ...}$^+$
NamedValue	→	`lower` [NamedValSuf]
	|	`upper` AuxVal1
	|	["-"] `number`
	|	AuxVal0
NamedValSuf	→	Value
	|	AuxVal2

SOUS-TYPES

SubtypeSpec	→	"(" {SubtypeValueSet "|" ...}$^+$ ")"
SubtypeValueSet	→	INCLUDES Type
	|	MIN SubValSetSuf
	|	FROM SubtypeSpec
	|	SIZE SubtypeSpec
	|	WITH InnerTypeSuf
	|	SVSAux
SubValSetSuf	→	["<"] ".." ["<"] UpperEndValue
UpperEndValue	→	Value
	|	MAX
InnerTypeSuf	→	COMPONENT SubtypeSpec
	|	COMPONENTS MultipleTypeConstraints
MultipleTypeConstraints	→	"{" ["..." ","] {[NamedConstraint] "," ...} "}"
NamedConstraint	→	`lower`$_{id}$ [SubtypeSpec] [PresenceConstraint]
	|	SubtypeSpec [PresenceConstraint]
	|	PresenceConstraint
PresenceConstraint	→	PRESENT
	|	ABSENT
	|	OPTIONAL
SVSAux	→	BuiltInValue [SubValSetSuf]
	|	AuxType " :" SVSAux
	|	NULL [SVSAux3]
	|	`upper` SVSAux1
	|	`lower` [SVSAux2]
	|	["-"] `number` [SubValSetSuf]
SVSAux1	→	SubtypeSpec* " :" SVSAux
	|	"." SVSAux11
SVSAux2	→	" :" SVSAux
	|	".." ["<"] UpperEndValue
	|	"<" SVSAux21
SVSAux3	→	SubtypeSpec* " :" SVSAux
	|	SubValSetSuf
SVSAux11	→	`upper`$_{typ}$ SubtypeSpec* " :" SVSAux
	|	`lower`$_{val}$ [SubValSetSuf]
SVSAux21	→	Type " :" SVSAux
	|	".." ["<"] UpperEndValue

10.3 Vérification de la propriété LL(1)

Nous apportons ici la preuve que la grammaire précédente obtenue après moult transformations est LL(1). Nous rappelons préalablement la définition exacte de la propriété LL(1).

10.3.1 Définition de la propriété LL(1)

Les grammaires LL(1) sont les grammaires analysables de façon descendante avec un lexème de prévision, sans reprise arrière (*Left to right scanning of the input, building a Leftmost derivation with one token of look-ahead*). Pour les définir formellement, il est nécessaire d'introduire au préalable deux fonctions. Nous noterons N l'ensemble des non-terminaux et $\mathcal{S}$ l'ensemble des terminaux.

La fonction *Premiers*

La fonction $\mathcal{P}$ (premiers) est la suivante :

$$\forall \mathrm{A} \in N.\mathcal{P}(\mathrm{A}) = \{x \in \mathcal{S} \cup \{\varepsilon\} \mid \mathrm{A} \overset{*}{\Longrightarrow} x\alpha \quad \text{et} \quad x \neq \varepsilon \quad \text{ou} \quad \mathrm{A} \overset{*}{\Longrightarrow} x\}$$

La fonction *Suivants*

La fonction $\mathcal{S}$ (suivants) est définie par :

$$\forall \mathrm{A} \in N.\mathcal{S}(\mathrm{A}) = \{x \in \mathcal{S} \mid \exists \mathrm{B} \in N, \mathrm{B} \overset{*}{\Longrightarrow} \alpha \mathrm{A} x \beta \quad \text{ou} \quad \mathrm{B} \overset{*}{\Longrightarrow} \alpha \mathrm{A} x\}$$

Définition LL(1)

Nous notons ici l'implication par $\models$, pour éviter toute confusion avec la relation $\Longrightarrow$. Une grammaire est LL(1) si et seulement si elle vérifie les propriétés suivantes :

$$\forall \mathrm{A} \in N.\neg(\mathrm{A} \overset{*}{\Longrightarrow} \mathrm{A}\alpha) \tag{P_1}$$

$$\mathrm{A} \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n \quad \models \quad \bigcap_{i=1}^{n} \mathcal{P}(\alpha_i) = \varnothing \tag{P_2}$$

$$\mathrm{A} \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n \quad \text{et} \quad \alpha_1 \overset{*}{\Longrightarrow} \varepsilon \quad \models \quad \forall i \in [1, n].\mathcal{P}(\alpha_i) \cap \mathcal{S}(\mathrm{A}) = \varnothing \tag{P_3}$$

Extension aux opérateurs rationnels

Nous avons vu au chapitre 10.1.1 page 234 la définition des opérateurs rationnels utilisés dans cette grammaire ASN.1. Nous indiquions

que nous pouvions considérer les expressions rationnelles comme étant produites par une règle spécifique. Nous allons étendre ici les fonctions $\mathcal{P}$ et $\mathcal{S}$ à ces expressions et en donner une définition récursive et algorithmique. Nouvelle définition de $\mathcal{P}$:

$$\begin{aligned}
\mathcal{P}(\varepsilon) &= \{\varepsilon\} \\
\mathcal{P}(x\gamma) &= \{x\} \\
\mathcal{P}(\mathrm{B}\gamma) &= \mathcal{P}(\mathrm{B}) \\
\mathcal{P}([\beta]\gamma) &= \mathcal{P}(\beta) \cup \mathcal{P}(\gamma) \\
\mathcal{P}(\{\mathrm{B}\,\mathrm{b}\ldots\}^*\gamma) &= \mathcal{P}(\mathrm{B}) \cup \mathcal{P}(\gamma) \\
\mathcal{P}(\{\mathrm{B}\,\mathrm{b}\ldots\}^+\gamma) &= \mathcal{P}(\mathrm{B}) \\
\mathcal{P}(\beta^*\gamma) &= \mathcal{P}(\beta) \cup \mathcal{P}(\gamma) \\
\mathcal{P}(\beta^+\gamma) &= \mathcal{P}(\beta) \\
\mathcal{P}(\mathrm{A}) &= \bigcup_{i=1}^{n} \mathcal{P}(\alpha_i),\ \text{si } \mathrm{A} \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n
\end{aligned}$$

Nous signalons que, pour faciliter la définition, nous avons étendu $\mathcal{P}$ à ε (bien que ε n'apparaisse jamais explicitement dans la grammaire) et γ peut valoir ε. Nouvelle définition de $\mathcal{S}$:

— $\mathrm{X} \to \ldots \mid \alpha\mathrm{AB}\beta$
$\mathcal{S}(\mathrm{A}) = \mathcal{P}(\mathrm{B})$

— $\mathrm{X} \to \ldots \mid \alpha\mathrm{A}\{\mathrm{B}\,\mathrm{b}\ldots\}^+\beta$
Idem.

— $\mathrm{X} \to \ldots \mid \alpha\mathrm{A}[\beta]\gamma$
$$\mathcal{S}(\mathrm{A}) = \begin{cases} \text{si } \gamma \overset{*}{\Longrightarrow} \varepsilon & \text{alors } \mathcal{P}(\beta) \cup (\mathcal{P}(\gamma)\backslash\{\varepsilon\}) \cup \mathcal{S}(\mathrm{X}) \\ & \text{sinon } \mathcal{P}(\beta) \cup \mathcal{P}(\gamma) \end{cases}$$

— $\mathrm{X} \to \ldots \mid \alpha\mathrm{A}\beta^*\gamma$
Idem.

— $\mathrm{X} \to \ldots \mid \alpha\mathrm{A}$
$\mathcal{S}(\mathrm{A}) = \mathcal{S}(\mathrm{X})$

— $\mathrm{X} \to \ldots \mid \alpha\mathrm{A}x\beta$
$\mathcal{S}(\mathrm{A}) = \{x\}$

— $\mathrm{X} \to \ldots \mid \alpha\mathrm{A}\beta^+\gamma$
$\mathcal{S}(\mathrm{A}) = \mathcal{P}(\beta)$

— $\mathrm{X} \to \ldots \mid \alpha\mathrm{A}\{\mathrm{Bb}\ldots\}^*\beta$
$$\mathcal{S}(\mathrm{A}) = \begin{cases} \text{si } \beta \overset{*}{\Longrightarrow} \varepsilon & \text{alors } \mathcal{P}(\mathrm{B}) \cup (\mathcal{P}(\beta)\backslash\{\varepsilon\}) \cup \mathcal{S}(\mathrm{X}) \\ & \text{sinon } \mathcal{P}(\mathrm{B}) \cup \mathcal{P}(\beta) \end{cases}$$

— $\mathrm{X} \to \ldots \mid \alpha\{\mathrm{Aa}\ldots\}^*\beta$
$$\mathcal{S}(\mathrm{A}) = \begin{cases} \text{si } \beta \overset{*}{\Longrightarrow} \varepsilon & \text{alors } \{\mathrm{a}\} \cup (\mathcal{P}(\beta)\backslash\{\varepsilon\}) \cup \mathcal{S}(\mathrm{X}) \\ & \text{sinon } \{\mathrm{a}\} \cup \mathcal{P}(\beta) \end{cases}$$

— $\mathrm{X} \to \ldots \mid \alpha\{\mathrm{Aa}\ldots\}^+\beta$
Idem.

Partout où cela est autorisé, nous pouvons lire le tableau précédent en remplaçant A par [A], A^* ou A^+. Ceci était la définition générale, mais dans cette étude les grammaires ne possèdent pas de productions vides explicites (cf. 10.1.2 page 236). Nous pouvons alors remplacer l'équation P_3 par l'algorithme suivant : à chaque occurrence des expressions rationnelles α^*, α^+, $[\alpha]$, {A a ...}*, et {A a ...}$^+$, les considérer comme étant produites par une règle dédiée (sans partage) et vérifier les contraintes associées suivantes :

Règle rationnelle	Contrainte
$\mathrm{X} \to \alpha^*$	$\mathcal{P}(\alpha) \cap \mathcal{S}(X) = \varnothing$
$\mathrm{X} \to \alpha^+$	$\mathcal{P}(\alpha) \cap \mathcal{S}(X) = \varnothing$
$\mathrm{X} \to [\alpha]$	$\mathcal{P}(\alpha) \cap \mathcal{S}(X) = \varnothing$
$\mathrm{X} \to \{\mathrm{A\ a}\ldots\}^*$	$(\mathcal{P}(A) \cup \{a\}) \cap \mathcal{S}(X) = \varnothing$
$\mathrm{X} \to \{\mathrm{A\ a}\ldots\}^+$	$\{a\} \cap \mathcal{S}(X) = \varnothing$

Le fait de ne pas considérer de partage facilite le travail s'il est réalisé « à la main » : le coût est identique, mais la tâche aura été plus localisée à chaque étape. Un programme pourrait se passer de ce choix, bien entendu.

10.3.2 Une grammaire LL(1) d'ASN.1 :1990

Équation $\mathbf{P}_1$

Nous présentons dans le tableau ci-dessous les non-terminaux qui sont produits en tête de chaque production. Nous vérifions aisément, par fermeture transitive, qu'il n'y a pas de récursivité gauche qui apparaît.

Règle	Tête
ModuleDefinition	ModuleIdentifier
ModuleIdentifier	
ObjIdComponent	
TagDefault	
ModuleBody	Exports, Imports, Assignment
Exports	
Imports	
SymbolsFromModule	Symbol
Symbol	
Assignment	

Type	AuxType
AuxType	BuiltInType, SetSeq
SetSeq	
TypeSuf	SubtypeSpec
BuiltInType	
NamedType	AuxType
NamedNumber	
AuxNamedNum	
NamedBit	
ElementType	NamedType
ElementTypeSuf	
Class	
ClassNumber	
Value	AuxVal0
AuxVal0	BuiltInValue, AuxType
AuxVal1	SpecVal
AuxVal2	
AuxVal11	
SpecVal	SubtypeSpec
BuiltInValue	
BetBraces	AuxVal0
AuxBet1	AuxNamed, AuxVal2, AuxVal0
AuxBet2	SpecVal
AuxBet3	ObjIdComponent, AuxNamed
AuxBet11	ObjIdComponent, AuxVal2, AuxNamed
AuxBet21	
AuxNamed	
NamedValue	AuxVal0
NamedValSuf	Value, AuxVal2
SubtypeSpec	
SubtypeValueSet	SVSAux
SubValSetSuf	
UpperEndValue	Value
InnerTypeSuf	
MultipleTypeConstraints	
NamedConstraint	SubtypeSpec, PresenceConstraint
PresenceConstraint	
SVSAux	BuiltInValue, AuxType
SVSAux1	SubtypeSpec
SVSAux2	
SVSAux3	SubtypeSpec, SubValSetSuf

SVSAux11 SVSAux21	 Type

Équation P_2

Pour toutes les règles nous écrivons la contrainte imposée par l'équation P_2, puis calculons les ensembles $\mathcal{P}(\alpha_i)$ nécessaires pour les résoudre. Pour faciliter la lecture nous ne ferons pas figurer la contrainte si tous les α_i sont des terminaux (trivial). De même, nous réduirons directement les $\mathcal{P}(\alpha_i)$, où α_i est une expression rationnelle spéciale, à leur forme sans opérateur (Cf. 10.3.1 page 246).

Règle	Contrainte
ModuleDefinition ModuleIdentifier ObjIdComponent TagDefault ModuleBody Exports Imports SymbolsFromModule Symbol Assignment	
Type	$\{\texttt{lower}, \texttt{upper}, \text{NULL}\}$ $\cap\, \mathcal{P}(\text{AuxType}) = \varnothing$
AuxType	$\{\text{“[”}\} \cap \mathcal{P}(\text{BuiltInType}) \cap \mathcal{P}(\text{SetSeq}) = \varnothing$
SetSeq	
TypeSuf	$\mathcal{P}(\text{SubtypeSpec}) \cap \{\text{“\{”}, \text{SIZE}, \text{OF}\} = \varnothing$
BuiltInType	
NamedType	$\{\texttt{lower}, \texttt{upper}, \text{NULL}\}$ $\cap\, \mathcal{P}(\text{AuxType}) = \varnothing$
NamedNumber AuxNamedNum NamedBit	
ElementType	$\mathcal{P}(\text{NamedType}) \cap \{\text{COMPONENTS}\} = \varnothing$
ElementTypeSuf Class ClassNumber	
Value	$\mathcal{P}(\text{AuxVal0})$ $\cap\, \{\texttt{upper}, \texttt{lower}, \text{“-”}, \texttt{number}\} = \varnothing$

AuxVal0	$\mathcal{P}$(BuiltInValue) $\cap\, \mathcal{P}$(AuxType) $\cap$ {NULL} $= \varnothing$
AuxVal1	$\mathcal{P}$(SpecVal) $\cap$ {“.”} $= \varnothing$
AuxVal2	
AuxVal11	
SpecVal	
BuiltInValue	
BetBraces	$\mathcal{P}$(AuxVal0) $\cap$ {“-”, `lower`, `upper`, `number`} $= \varnothing$
AuxBet1	$\mathcal{P}$(AuxNamed) $\cap\, \mathcal{P}$(AuxVal2) $\cap\, \mathcal{P}$(AuxVal0) $\cap$ {“(”, “-”, `lower`, `upper`, `number`} $= \varnothing$
AuxBet2	$\mathcal{P}$(SpecVal) $\cap$ {“.”} $= \varnothing$
AuxBet3	$\mathcal{P}$(ObjIdComponent) $\cap\, \mathcal{P}$(AuxNamed) $= \varnothing$
AuxBet11	{“(”} $\cap\, \mathcal{P}$(ObjIdComponent) $\cap\, \mathcal{P}$(AuxVal2) $\cap\, \mathcal{P}$(AuxNamed) $= \varnothing$
AuxBet21	
AuxNamed	
NamedValue	$\mathcal{P}$(AuxVal0) $\cap$ {“-”, `lower`, `upper`, `number`} $= \varnothing$
NamedValSuf	$\mathcal{P}$(Value) $\cap\, \mathcal{P}$(AuxVal2) $= \varnothing$
SubtypeSpec	
SubtypeValueSet	{INCLUDES, MIN, FROM, SIZE, WITH} $\cap\, \mathcal{P}$(SVSAux) $= \varnothing$
SubValSetSuf	
UpperEndValue	$\mathcal{P}$(Value) $\cap$ {MAX} $= \varnothing$
InnerTypeSuf	
MultipleTypeConstraints	
NamedConstraint	{`lower`} $\cap\, \mathcal{P}$(SubtypeSpec) $\cap\, \mathcal{P}$(PresenceConstraint) $= \varnothing$
PresenceConstraint	
SVSAux	$\mathcal{P}$(BuiltInValue) $\cap\, \mathcal{P}$(AuxType) $\cap$ {NULL, `upper`} $\cap$ {`lower`, “-”, `number`} $= \varnothing$
SVSAux1	$\mathcal{P}$(SubtypeSpec) $\cap$ {“.”, “ :”} $= \varnothing$
SVSAux2	
SVSAux3	$\mathcal{P}$(SubtypeSpec) $\cap$ {“ :”} $\cap\, \mathcal{P}$(SubValSetSuf) $= \varnothing$
SVSAux11	
SVSAux21	$\mathcal{P}$(Type) $\cap$ {“..”} $= \varnothing$

Finalement, après simplification, le système d'équations suivant doit être satisfait :

(1) $\{\text{“[”}\} \cap \mathcal{P}(\text{BuiltInType}) \cap \mathcal{P}(\text{SetSeq}) = \varnothing$
(2) $\mathcal{P}(\text{SubtypeSpec}) \cap \{\text{“\{”}, \text{“.”}, \text{“ :”}, \textsc{size}, \textsc{of}\} = \varnothing$
(3) $\mathcal{P}(\text{NamedType}) \cap \{\textsc{components}\} = \varnothing$
(4) $\mathcal{P}(\text{SpecVal}) \cap \{\text{“.”}\} = \varnothing$
(5) $\mathcal{P}(\text{AuxNamed}) \cap \mathcal{P}(\text{AuxVal2}) \cap \mathcal{P}(\text{AuxVal0}) \cap \{\text{“(”}, \text{“-”}, \texttt{lower}, \texttt{upper}, \texttt{number}\} = \varnothing$
(6) $\{\text{“(”}\} \cap \mathcal{P}(\text{ObjIdComponent}) \cap \mathcal{P}(\text{AuxVal2}) \cap \mathcal{P}(\text{AuxNamed}) = \varnothing$
(7) $\mathcal{P}(\text{Value}) \cap \mathcal{P}(\text{AuxVal2}) = \varnothing$
(8) $\mathcal{P}(\text{Value}) \cap \{\textsc{max}\} = \varnothing$
(9) $\{\textsc{includes}, \textsc{min}, \textsc{from}, \textsc{size}, \textsc{with}\} \cap \mathcal{P}(\text{SVSAux}) = \varnothing$
(10) $\{\texttt{lower}\} \cap \mathcal{P}(\text{SubtypeSpec}) \cap \mathcal{P}(\text{PresenceConstraint}) = \varnothing$
(11) $\mathcal{P}(\text{BuiltInValue}) \cap \mathcal{P}(\text{AuxType}) \cap \{\textsc{null}, \texttt{upper}, \texttt{lower}, \text{“-”}, \texttt{number}\} = \varnothing$
(12) $\mathcal{P}(\text{SubtypeSpec}) \cap \{\text{“ :”}\} \cap \mathcal{P}(\text{SubValSetSuf}) = \varnothing$
(13) $\mathcal{P}(\text{Type}) \cap \{\text{“..”}\} = \varnothing$

Or :

$\mathcal{P}(\text{BuiltInType})$ = { BOOLEAN, INTEGER, BIT, OCTET, CHOICE, ANY, OBJECT, ENUMERATED, REAL, EXTERNAL, “NumericString”, “PrintableString”, “TeletexString”, “T61String”, “VideotexString”, “VisibleString”, “ISO646String”, “IA5String”, “GraphicString”, “GeneralString”, “UTCTime”, “GeneralizedTime”, “ObjectDescriptor” }

$\mathcal{P}(\text{SetSeq})$ = { SET, SEQUENCE }

Donc l'équation (1) est vérifiée.

$$\mathcal{P}(\text{SubtypeSpec}) = \{\text{“(”}\}$$

Donc l'équation (2) est vérifiée.

$$
\begin{array}{lcl}
\mathcal{P}(\text{NamedType}) & = & \{\texttt{lower}, \texttt{upper}, \text{NULL}\} \cup \mathcal{P}(\text{AuxType}) \\
\mathcal{P}(\text{AuxType}) & = & \{\text{“[”}\} \cup \mathcal{P}(\text{BuiltInType}) \cup \mathcal{P}(\text{SetSeq}) \\
 & = & \{\ \text{“[”, SET, SEQUENCE, BOOLEAN, INTEGER,} \\
 & & \text{BIT, OCTET, CHOICE, ANY, OBJECT,} \\
 & & \text{ENUMERATED, REAL,} \\
 & & \text{EXTERNAL, “NumericString”, “PrintableString”,} \\
 & & \text{“TeletexString”, “T61String”, “VideotexString”,} \\
 & & \text{“VisibleString”, “ISO646String”, “IA5String”,} \\
 & & \text{“GraphicString”, “GeneralString”, “UTCTime”,} \\
 & & \text{“GeneralizedTime”, “ObjectDescriptor”}\ \}
\end{array}
$$

Donc l'équation (3) est vérifiée.

$$
\begin{array}{lcl}
\mathcal{P}(\text{SpecVal}) & = & \mathcal{P}(\text{SubtypeSpec}) \cup \{\text{“ :”}\} \\
 & = & \{\ \text{“(”, “ :”}\ \}
\end{array}
$$

Donc l'équation (4) est vérifiée.

$$
\begin{array}{lcl}
\mathcal{P}(\text{AuxNamed}) & = & \{\ \text{“,”}\ \} \\
\mathcal{P}(\text{AuxVal2}) & = & \{\ \text{“<”, “ :”}\ \} \\
\mathcal{P}(\text{BuiltInValue}) & = & \{\ \text{TRUE, FALSE, PLUS-INFINITY, MINUS-INFINITY,} \\
 & & \texttt{basednum}, \texttt{string}, \text{“\{”}\ \} \\
\mathcal{P}(\text{AuxVal0}) & = & \mathcal{P}(\text{BuiltInValue}) \cup \mathcal{P}(\text{AuxType}) \cup \{\text{NULL}\} \\
 & = & \{\ \text{TRUE, FALSE, PLUS-INFINITY, MINUS-INFINITY,} \\
 & & \texttt{basednum}, \texttt{string}, \text{“\{”, NULL, “[”, SET, SEQUENCE,} \\
 & & \text{BOOLEAN, INTEGER, BIT, OCTET, CHOICE, ANY,} \\
 & & \text{OBJECT, ENUMERATED, REAL, EXTERNAL,} \\
 & & \text{“NumericString”, “PrintableString”, “TeletexString”,} \\
 & & \text{“T61String”, “VideotexString”, “VisibleString”,} \\
 & & \text{“ISO646String”, “IA5String”, “GraphicString”,} \\
 & & \text{“GeneralString”, “UTCTime”, “GeneralizedTime”,} \\
 & & \text{“ObjectDescriptor”}\ \}
\end{array}
$$

Donc les équations (5) et (11) sont vérifiées.

$$\mathcal{P}(\text{ObjIdComponent}) = \{\texttt{number}, \texttt{upper}, \texttt{lower}\}$$

Donc l'équation (6) est vérifiée.

$$
\begin{aligned}
\mathcal{P}(\text{Value}) &= \mathcal{P}(\text{AuxVal0}) \cup \{\texttt{upper}, \texttt{lower}, \texttt{number}, \text{“-”}\} \\
&= \{\ \text{TRUE, FALSE, PLUS-INFINITY,} \\
&\quad \text{MINUS-INFINITY, } \texttt{basednum}, \\
&\quad \texttt{string}, \text{“\{”, NULL, “[”, SET, SEQUENCE, BOOLEAN,} \\
&\quad \text{INTEGER, BIT, OCTET, CHOICE, ANY, OBJECT,} \\
&\quad \text{ENUMERATED, REAL, EXTERNAL,} \\
&\quad \text{“NumericString”, “PrintableString”, “TeletexString”,} \\
&\quad \text{“T61String”, “VideotexString”, “VisibleString”,} \\
&\quad \text{“ISO646String”, “IA5String”, “GraphicString”,} \\
&\quad \text{“GeneralString”, “UTCTime”, “GeneralizedTime”,} \\
&\quad \text{“ObjectDescriptor”, } \texttt{upper}, \texttt{lower}, \texttt{number}, \text{“-”}\ \}
\end{aligned}
$$

Donc les équations (7) et (8) sont vérifiées.

$$
\begin{aligned}
\mathcal{P}(\text{SVSAux}) &= \mathcal{P}(\text{BuiltInValue}) \cup \mathcal{P}(\text{Auxtype}) \\
&\quad \cup\ \{\ \text{NULL, } \texttt{upper}, \texttt{lower}, \texttt{number}, \text{“-”}\ \} \\
&= \{\ \text{TRUE, FALSE, PLUS-INFINITY, MINUS-INFINITY,} \\
&\quad \texttt{basednum}, \texttt{string}, \text{“\{”, “[”, SET, SEQUENCE, BOOLEAN,} \\
&\quad \text{INTEGER, BIT, OCTET, CHOICE, ANY, OBJECT,} \\
&\quad \text{ENUMERATED, REAL, EXTERNAL, “NumericString”,} \\
&\quad \text{“PrintableString”, “TeletexString”, “T61String”,} \\
&\quad \text{“VideotexString”, “VisibleString”, “ISO646String”,} \\
&\quad \text{“IA5String”, “GraphicString”, “GeneralString”,} \\
&\quad \text{“UTCTime”, “GeneralizedTime”,“ObjectDescriptor”,} \\
&\quad \text{NULL, } \texttt{upper}, \texttt{lower}, \texttt{number}, \text{“-”}\ \}
\end{aligned}
$$

Donc l'équation (9) est vérifiée.

$$\mathcal{P}(\text{PresenceConstraint}) = \{\text{PRESENT, ABSENT, OPTIONAL}\}$$

Donc l'équation (10) est vérifiée.

$$\mathcal{P}(\text{SubValSetSuf}) = \{\text{“<”, “..”}\}$$

Donc l'équation (12) est vérifiée.

$$
\begin{array}{rcl}
\mathcal{P}(\text{Type}) & = & \{\texttt{lower}, \texttt{upper}, \text{NULL}\} \cup \mathcal{P}(\text{AuxType}) \\
& = & \{\ \texttt{lower}, \texttt{upper}, \text{NULL}, \\
& & \text{“[”, SET, SEQUENCE, BOOLEAN, INTEGER, BIT,} \\
& & \text{OCTET, CHOICE, ANY, OBJECT,} \\
& & \text{ENUMERATED, REAL, EXTERNAL,} \\
& & \text{“NumericString”, “PrintableString”, “TeletexString”,} \\
& & \text{“T61String”, “VideotexString”, “VisibleString”,} \\
& & \text{“ISO646String”, “IA5String”,“GraphicString”,} \\
& & \text{“GeneralString”, “UTCTime”, “GeneralizedTime”,} \\
& & \text{“ObjectDescriptor”}\ \}
\end{array}
$$

Donc l'équation (13) est vérifiée.

Équation P_3

Nous donnons ici pour chaque production de chaque règle les contraintes imposées par l'équation P_3 (cf. 10.3.1 page 246). Pour ne pas charger inutilement le tableau, les équations redondantes au sein d'une même règle et les équations triviales n'apparaîtront pas. Nous simplifierons de même ce qui peut l'être localement.

Règle	Contraintes
ModuleDefinition	$\mathcal{P}(\text{TagDefault}) \cap \{\text{“ : :=”}\} = \varnothing$ $\mathcal{P}(\text{ModuleBody}) \cap \{\text{END}\} = \varnothing$
ModuleIdentifier	$\{\text{“\{”}\} \cap \mathcal{S}(\text{ModuleIdentifier}) = \varnothing$
ObjIdComponent	$\{\text{“(”}\} \cap \mathcal{S}(\text{ObjIdComponent}) = \varnothing$
TagDefault	
ModuleBody	$\mathcal{P}(\text{Exports}) \cap (\mathcal{P}(\text{Imports}) \cup \mathcal{P}(\text{Assignment})) = \varnothing$ $\mathcal{P}(\text{Imports}) \cap \mathcal{P}(\text{Assignment}) = \varnothing$
Exports	$(\mathcal{P}(\text{Symbol}) \cup \{\text{“,”}\}) \cap \{\text{“ ;”}\} = \varnothing$
Imports	$\mathcal{P}(\text{SymbolsFromModule}) \cap \{\text{“ ;”}\} = \varnothing$
SymbolsFromModule	
Symbol	
Assignment	
Type	$\{\text{“.”}\} \cap (\mathcal{P}(\text{SubtypeSpec}) \cup \mathcal{S}(\text{Type})) = \varnothing$ $\mathcal{P}(\text{SubtypeSpec}) \cap \mathcal{S}(\text{Type}) = \varnothing$
AuxType	$\mathcal{P}(\text{Class}) \cap \mathcal{P}(\text{ClassNumber}) = \varnothing$ $\mathcal{P}(\text{TagDefault}) \cap \mathcal{P}(\text{Type}) = \varnothing$ $\mathcal{P}(\text{TypeSuf}) \cap \mathcal{S}(\text{AuxType}) = \varnothing$
SetSeq	
TypeSuf	$\mathcal{P}(\text{SubtypeSpec}) \cap \mathcal{S}(\text{TypeSuf}) = \varnothing$ $(\mathcal{P}(\text{ElementType}) \cup \{\text{“,”}\}) \cap \{\text{“}\}\text{”}\} = \varnothing$

BuiltInType	$\{\text{“}\{\text{”}\} \cap \mathcal{S}(\text{BuiltInType}) = \varnothing$ $\{\text{DEFINED}\} \cap \mathcal{S}(\text{BuiltInType}) = \varnothing$
NamedType	$\{\text{“}<\text{”}\} \cap \mathcal{P}(\text{Type}) = \varnothing$ $\{\text{“.”}\} \cap (\mathcal{P}(\text{SubtypeSpec}) \cup \mathcal{S}(\text{NamedType})) = \varnothing$ $\mathcal{P}(\text{SubtypeSpec}) \cap \mathcal{S}(\text{NamedType}) = \varnothing$
NamedNumber	
AuxNamedNum	
NamedBit	
ElementType	$\mathcal{P}(\text{ElementTypeSuf}) \cap \mathcal{S}(\text{ElementType}) = \varnothing$
ElementTypeSuf	
Class	
ClassNumber	
Value	$\mathcal{P}(\text{AuxVal2}) \cap \mathcal{S}(\text{Value}) = \varnothing$
AuxVal0	$\mathcal{P}(\text{SpecVal}) \cap \mathcal{S}(\text{AuxVal0}) = \varnothing$
AuxVal1	
AuxVal2	
AuxVal11	
SpecVal	$\mathcal{P}(\text{SubtypeSpec}) \cap \{\text{“ :”}\} = \varnothing$
BuiltInValue	
BetBraces	$\mathcal{P}(\text{AuxNamed}) \cap \mathcal{S}(\text{BetBraces}) = \varnothing$ $\mathcal{P}(\text{AuxBet1}) \cap \mathcal{S}(\text{BetBraces}) = \varnothing$ $\mathcal{P}(\text{AuxBet3}) \cap \mathcal{S}(\text{BetBraces}) = \varnothing$
AuxBet1	$\mathcal{P}(\text{ObjIdComponent}) \cap \mathcal{S}(\text{AuxBet1}) = \varnothing$ $\mathcal{P}(\text{AuxNamed}) \cap \mathcal{S}(\text{AuxBet1}) = \varnothing$ $\mathcal{P}(\text{AuxBet11}) \cap \mathcal{S}(\text{AuxBet1}) = \varnothing$ $\mathcal{P}(\text{AuxBet3}) \cap \mathcal{S}(\text{AuxBet1}) = \varnothing$
AuxBet2	$\mathcal{P}(\text{AuxNamed}) \cap \mathcal{S}(\text{AuxBet2}) = \varnothing$
AuxBet3	$\mathcal{P}(\text{ObjIdComponent}) \cap \mathcal{S}(\text{AuxBet3}) = \varnothing$
AuxBet11	$\mathcal{P}(\text{ObjIdComponent}) \cap \mathcal{S}(\text{AuxBet11}) = \varnothing$ $\mathcal{P}(\text{AuxNamed}) \cap \mathcal{S}(\text{AuxBet11}) = \varnothing$
AuxBet21	$\mathcal{P}(\text{AuxNamed}) \cap \mathcal{S}(\text{AuxBet21}) = \varnothing$ $\mathcal{P}(\text{AuxBet3}) \cap \mathcal{S}(\text{AuxBet21}) = \varnothing$
AuxNamed	$\{\text{“,”}\} \cap \mathcal{S}(\text{AuxNamed}) = \varnothing$
NamedValue	$\mathcal{P}(\text{NamedValSuf}) \cap \mathcal{S}(\text{NamedValue}) = \varnothing$
NamedValSuf	
SubtypeSpec	
SubtypeValueSet	
SubValSetSuf	$\{\text{“}<\text{”}\} \cap \mathcal{P}(\text{UpperEndValue}) = \varnothing$
UpperEndValue	

InnerTypeSuf	
MultipleTypeConstraints	$\{\text{“...”}, \text{“}\}\text{”}\} \cap \mathcal{P}(\text{NamedConstraint}) = \varnothing$
NamedConstraint	$\mathcal{P}(\text{SubtypeSpec}) \cap (\mathcal{P}(\text{PresenceConstraint}) \cup \mathcal{S}(\text{NamedConstraint})) = \varnothing$ $\mathcal{P}(\text{PresenceConstraint}) \cap \mathcal{S}(\text{NamedConstraint}) = \varnothing$
PresenceConstraint	
SVSAux	$\mathcal{P}(\text{SubValSetSuf}) \cap \mathcal{S}(\text{SVSAux}) = \varnothing$ $\mathcal{P}(\text{SVSAux3}) \cap \mathcal{S}(\text{SVSAux}) = \varnothing$ $\mathcal{P}(\text{SVSAux2}) \cap \mathcal{S}(\text{SVSAux}) = \varnothing$
SVSAux1	$\mathcal{P}(\text{SubtypeSpec}) \cap \{\text{“ :”}\} = \varnothing$
SVSAux2	$\{\text{“<”}\} \cap \mathcal{P}(\text{UpperEndValue}) = \varnothing$
SVSAux3	$\mathcal{P}(\text{SubtypeSpec}) \cap \{\text{“ :”}\} = \varnothing$
SVSAux11	$\mathcal{P}(\text{SubtypeSpec}) \cap \{\text{“ :”}\} = \varnothing$ $\mathcal{P}(\text{SubValSetSuf}) \cap \mathcal{S}(\text{SVSAux11}) = \varnothing$
SVSAux21	$\{\text{“<”}\} \cap \mathcal{P}(\text{UpperEndValue}) = \varnothing$

Finalement, après simplifications, le système d'équations suivant doit être satisfait :

(1) $\mathcal{P}(\text{TagDefault}) \cap \{\text{“ : :=”}\} = \varnothing$
(2) $\mathcal{P}(\text{ModuleBody}) \cap \{\text{END}\} = \varnothing$
(3) $\{\text{“\{”}\} \cap \mathcal{S}(\text{ModuleIdentifier}) = \varnothing$
(4) $\{\text{“(”}\} \cap \mathcal{S}(\text{ObjIdComponent}) = \varnothing$
(5) $\mathcal{P}(\text{Exports}) \cap \mathcal{P}(\text{Imports}) = \varnothing$
(6) $\mathcal{P}(\text{Exports}) \cap \mathcal{P}(\text{Assignment}) = \varnothing$
(7) $\mathcal{P}(\text{Imports}) \cap \mathcal{P}(\text{Assignment}) = \varnothing$
(8) $\mathcal{P}(\text{TypeSuf}) \cap \mathcal{S}(\text{AuxType}) = \varnothing$
(9) $\mathcal{P}(\text{SymbolsFromModule}) \cap \{\text{“;”}\} = \varnothing$
(10) $\mathcal{P}(\text{SubtypeSpec}) \cap \{\text{“.”, “ :”}\} = \varnothing$
(11) $\mathcal{S}(\text{Type}) \cap \{\text{“.”}\} = \varnothing$
(12) $\mathcal{P}(\text{Class}) \cap \mathcal{P}(\text{ClassNumber}) = \varnothing$
(13) $\mathcal{P}(\text{TagDefault}) \cap \mathcal{P}(\text{Type}) = \varnothing$
(14) $\mathcal{P}(\text{Type}) \cap \{\text{“<”}\} = \varnothing$
(15) $\mathcal{P}(\text{SubtypeSpec}) \cap \mathcal{S}(\text{TypeSuf}) = \varnothing$
(16) $\mathcal{P}(\text{ElementType}) \cap \{\text{“}\}\text{”}\} = \varnothing$
(17) $\{\text{“\{”, DEFINED}\} \cap \mathcal{S}(\text{BuiltInType}) = \varnothing$
(18) $\{\text{“.”}\} \cap \mathcal{S}(\text{NamedType}) = \varnothing$
(19) $\mathcal{P}(\text{SubtypeSpec}) \cap \mathcal{S}(\text{NamedType}) = \varnothing$

(20) $\mathcal{P}(\text{ElementTypeSuf}) \cap \mathcal{S}(\text{ElementType}) = \varnothing$
(21) $\mathcal{P}(\text{AuxVal2}) \cap \mathcal{S}(\text{Value}) = \varnothing$
(22) $\mathcal{P}(\text{SpecVal}) \cap \mathcal{S}(\text{AuxVal0}) = \varnothing$
(23) $\mathcal{P}(\text{AuxNamed}) \cap \mathcal{S}(\text{BetBraces}) = \varnothing$
(24) $\mathcal{P}(\text{AuxBet1}) \cap \mathcal{S}(\text{BetBraces}) = \varnothing$
(25) $\mathcal{P}(\text{AuxBet3}) \cap \mathcal{S}(\text{BetBraces}) = \varnothing$

(26) $\mathcal{P}(\text{ObjIdComponent}) \cap \mathcal{S}(\text{AuxBet1}) = \varnothing$
(27) $\mathcal{P}(\text{AuxNamed}) \cap \mathcal{S}(\text{AuxBet1}) = \varnothing$
(28) $\mathcal{P}(\text{AuxBet11}) \cap \mathcal{S}(\text{AuxBet1}) = \varnothing$
(29) $\mathcal{P}(\text{AuxBet3}) \cap \mathcal{S}(\text{AuxBet1}) = \varnothing$
(30) $\mathcal{P}(\text{AuxNamed}) \cap \mathcal{S}(\text{AuxBet2}) = \varnothing$
(31) $\mathcal{P}(\text{ObjIdComponent}) \cap \mathcal{S}(\text{AuxBet3}) = \varnothing$
(32) $\mathcal{P}(\text{ObjIdComponent}) \cap \mathcal{S}(\text{AuxBet11}) = \varnothing$
(33) $\mathcal{P}(\text{AuxNamed}) \cap \mathcal{S}(\text{AuxBet11}) = \varnothing$
(34) $\mathcal{P}(\text{AuxNamed}) \cap \mathcal{S}(\text{AuxBet21}) = \varnothing$
(35) $\mathcal{P}(\text{AuxBet3}) \cap \mathcal{S}(\text{AuxBet21}) = \varnothing$
(36) $\{\text{“,”}\} \cap \mathcal{S}(\text{AuxNamed}) = \varnothing$
(37) $\mathcal{P}(\text{NamedValSuf}) \cap \mathcal{S}(\text{NamedValue}) = \varnothing$
(38) $\{\text{“<”}\} \cap \mathcal{P}(\text{UpperEndValue}) = \varnothing$
(39) $\{\text{“...”}, \text{“}\}\text{”}\} \cap \mathcal{P}(\text{NamedConstraint}) = \varnothing$
(40) $\mathcal{P}(\text{SubtypeSpec}) \cap \mathcal{P}(\text{PresenceConstraint}) = \varnothing$
(41) $\mathcal{P}(\text{SubtypeSpec}) \cap \mathcal{S}(\text{NamedConstraint}) = \varnothing$
(42) $\mathcal{P}(\text{PresenceConstraint}) \cap \mathcal{S}(\text{NamedConstraint}) = \varnothing$
(43) $\mathcal{P}(\text{SubValSetSuf}) \cap \mathcal{S}(\text{SVSAux}) = \varnothing$
(44) $\mathcal{P}(\text{SVSAux3}) \cap \mathcal{S}(\text{SVSAux}) = \varnothing$
(45) $\mathcal{P}(\text{SVSAux2}) \cap \mathcal{S}(\text{SVSAux}) = \varnothing$
(46) $\mathcal{P}(\text{SubValSetSuf}) \cap \mathcal{S}(\text{SVSAux11}) = \varnothing$
(47) $\mathcal{P}(\text{SubtypeSpec}) \cap \mathcal{S}(\text{Type}) = \varnothing$

Les équations (10), (14) et (40) sont immédiatement vérifiées à l'aide des ensembles $\mathcal{P}$ précédemment calculés (cf. 10.3.2 page 250). Nous calculons alors les ensembles $\mathcal{P}$ manquant :

$$\mathcal{P}(\text{TagDefault}) = \{\text{EXPLICIT, IMPLICIT}\}$$

Donc les équations (1) et (13) sont vérifiées.

$\mathcal{P}$(Exports) = {EXPORTS}
$\mathcal{P}$(Imports) = {IMPORTS}

Donc l'équation (5) est vérifiée.

$\mathcal{P}$(Assignment) = {`upper`, `lower`}

Donc les équations (6) et (7) sont vérifiées.

$$\begin{array}{lcl} \mathcal{P}(\text{ModuleBody}) & = & \mathcal{P}(\text{Exports}) \cup \mathcal{P}(\text{Imports}) \cup \mathcal{P}(\text{Assignment}) \\ & = & \{\text{EXPORTS, IMPORTS, } \texttt{upper}, \texttt{lower}\} \end{array}$$

Donc l'équation (2) est vérifiée.

$$\begin{array}{lcl} \mathcal{P}(\text{Symbol}) & = & \{ \texttt{upper}, \texttt{lower} \} \\ \mathcal{P}(\text{SymbolsFromModule}) & = & \mathcal{P}(\text{Symbol}) \cup \{\text{“ ;”}\} \\ & = & \{ \texttt{upper}, \texttt{lower}, \text{“ ;” } \} \end{array}$$

Donc l'équation (9) est vérifiée.

$$\begin{array}{lcl} \mathcal{P}(\text{Class}) & = & \{ \text{UNIVERSAL, APPLICATION, PRIVATE } \} \\ \mathcal{P}(\text{ClassNumber}) & = & \{ \texttt{number}, \texttt{upper}, \texttt{lower} \} \end{array}$$

Donc l'équation (12) est vérifiée.

$\mathcal{P}$(ElementType) = $\mathcal{P}$(NamedType) $\cup$ {COMPONENTS}

Donc l'équation (16) est vérifiée.

$\mathcal{P}$(UpperEndValue) = $\mathcal{P}$(Value) $\cup \{MAX\}$

Donc l'équation (38) est vérifiée.

$$\begin{array}{lcl} \mathcal{P}(\text{SubtypeSpec}) & = & \{ \text{“(” } \} \\ \mathcal{P}(\text{NamedConstraint}) & = & \{\texttt{lower}\} \cup \mathcal{P}(\text{SubtypeSpec}) \cup \mathcal{P}(\text{PresenceConstraint}) \\ & = & \{ \texttt{lower}, \text{“(”, PRESENT, ABSENT, OPTIONAL } \} \end{array}$$

Donc l'équation (39) est vérifiée.

De plus :

$\mathcal{P}$(TypeSuf) = $\mathcal{P}$(SubtypeSpec) $\cup$ {“{”, SIZE, OF}
= { “(”, “{”, SIZE, OF }

$\mathcal{P}$(ElementTypeSuf) = { OPTIONAL, DEFAULT }

$\mathcal{P}$(AuxVal2) = { “<”, “ :” }

$\mathcal{P}$(SpecVal) = $\mathcal{P}$(SubtypeSpec) $\cup$ {“ :”}
= { “(”, “ :” }

$\mathcal{P}$(AuxBet1) = {“(”, “-”, `upper`, `lower`, `number`} $\cup$ $\mathcal{P}$(AuxNamed)
$\cup$ $\mathcal{P}$(AuxVal2) $\cup$ $\mathcal{P}$(AuxVal0)
= { “(”, “-”, `upper`, `lower`, `number`, “,”, “<”, “ :”,
TRUE, FALSE, PLUS-INFINITY, MINUS-INFINITY,
`basednum`, `string`, “{”, NULL, “[”, SET, SEQUENCE,
BOOLEAN, INTEGER, BIT, OCTET, CHOICE,
ANY, OBJECT, ENUMERATED, REAL, EXTERNAL,
“NumericString”, “PrintableString”, “TeletexString”,
“T61String”, “VideotexString”, “VisibleString”,
“ISO646String”, “IA5String”, “GraphicString”,
“GeneralString”, “UTCTime”, “GeneralizedTime”,
“ObjectDescriptor” }

$\mathcal{P}$(AuxBet3) = $\mathcal{P}$(ObjIdComponent) $\cup$ $\mathcal{P}$(AuxNamed)
= { `number`, `upper`, `lower`, “,” }

$\mathcal{P}$(NamedValSuf) = $\mathcal{P}$(Value) $\cup$ $\mathcal{P}$(AuxVal2)
= { TRUE, FALSE, PLUS-INFINITY, MINUS-INFINITY,
`basednum`, `string`, “{”, NULL, “[”, SET, SEQUENCE,
BOOLEAN, INTEGER, BIT, OCTET, CHOICE,
ANY, OBJECT, ENUMERATED, REAL, EXTERNAL,
“NumericString”, “PrintableString”, “TeletexString”,
“T61String”, “VideotexString”, “VisibleString”,
“ISO646String”, “IA5String”, “GraphicString”,
“GeneralString”, “UTCTime”, “GeneralizedTime”,
“ObjectDescriptor”, `upper`, `lower`, `number`, “-”, “<”, “ :” }

$$\begin{array}{lcl}
\mathcal{P}(\text{AuxBet11}) & = & \{\text{“(”}\} \cup \mathcal{P}(\text{ObjIdComponent}) \cup \mathcal{P}(\text{AuxVal2}) \\
 & & \cup\, \mathcal{P}(\text{AuxNamed}) \\
 & = & \{\ \text{“(”, } \texttt{number}\text{, } \texttt{upper}\text{, } \texttt{lower}\text{, “<”, “ :”, “,” }\ \} \\
 & & \\
\mathcal{P}(\text{SubValSetSuf}) & = & \{\ \text{“<”, “..” }\ \} \\
 & & \\
\mathcal{P}(\text{SVSAux3}) & = & \mathcal{P}(\text{SubtypeSpec}) \cup \mathcal{P}(\text{SubValSetSuf}) \cup \{\text{“ :”}\} \\
 & = & \{\ \text{“(”, “<”, “..”, “ :” }\ \} \\
 & & \\
\mathcal{P}(\text{SVSAux2}) & = & \{\ \text{“ :”, “..”, “<” }\ \}
\end{array}$$

Donc $\mathcal{P}(\text{SubValSetSuf}) \subset \mathcal{P}(\text{SVSAux2}) \subset \mathcal{P}(\text{SVSAux3})$,
ce qui permet de supprimer les équations (43) et (45), qui sont impliquées par (44).

De plus, il est pertinent de noter ici :

$$\begin{array}{lcl}
\mathcal{S}(\text{SVSAux11}) & = & \mathcal{S}(\text{SVSAux1}) \\
\mathcal{S}(\text{SVSAux1}) & = & \mathcal{S}(\text{SVSAux})
\end{array}$$

Donc nous pouvons supprimer l'équation (46) car elle est impliquée par (44).

Nous calculons maintenant quelques ensembles $\mathcal{S}$ qui, à l'aide des ensembles $\mathcal{P}$ précédemment calculés, nous permettent de conclure en une étape.

$$\begin{array}{lcl}
\mathcal{S}(\text{SymbolsFromModule}) & = & \mathcal{P}(\text{SymbolsFromModule}) \cup \{\text{“ ;”}\} \\
 & = & \{\ \texttt{upper}\text{, } \texttt{lower}\text{, “ ;” }\ \} \\
\mathcal{S}(\text{ModuleIdentifier}) & = & \{\text{DEFINITIONS}\} \cup \mathcal{S}(\text{SymbolsFromModule}) \\
 & = & \{\ \text{DEFINITIONS, } \texttt{upper}\text{, } \texttt{lower}\text{, “ ;” }\ \}
\end{array}$$

Donc l'équation (3) est vérifiée.

$$\mathcal{S}(\text{ElementType}) = \{\text{“,”, “}\}\text{”}\}$$

Donc l'équation (20) est vérifiée.

$$\begin{array}{lcl}
\mathcal{S}(\text{NamedType}) & = & \{\text{“,”, “}\}\text{”}\} \cup \mathcal{P}(\text{ElementTypeSuf}) \cup \mathcal{S}(\text{ElementType}) \\
 & = & \{\ \text{“,”, “}\}\text{”, OPTIONAL, DEFAULT }\ \}
\end{array}$$

Donc les équations (18) et (19) sont vérifiées.

$\mathcal{S}(\text{BetBraces}) = \{\text{“}\}\text{”}\}$

Donc les équations (23), (24) et (25) sont vérifiées

$$\begin{array}{rcl} \mathcal{S}(\text{AuxBet1}) & = & \mathcal{S}(\text{BetBraces}) \\ & = & \{\ \text{“}\}\text{”}\ \} \end{array}$$

Donc les équations (26), (27), (28), et (29) sont vérifiées.

$$\begin{array}{rcl} \mathcal{S}(\text{AuxBet11}) & = & \mathcal{S}(\text{AuxBet1}) \\ & = & \{\ \text{“}\}\text{”}\ \} \end{array}$$

Donc les équations (32) et (33) sont vérifiées.

$$\begin{array}{rcl} \mathcal{S}(\text{AuxBet2}) & = & \mathcal{S}(\text{BetBraces}) \cup \mathcal{S}(\text{AuxBet1}) \\ & = & \{\ \text{“}\}\text{”}\ \} \end{array}$$

Donc l'équation (30) est vérifiée.

$$\begin{array}{rcl} \mathcal{S}(\text{AuxBet21}) & = & \mathcal{S}(\text{AuxBet2}) \\ & = & \{\ \text{“}\}\text{”}\ \} \end{array}$$

Donc les équations (34) et (35) sont vérifiées.

$$\begin{array}{rcl} \mathcal{S}(\text{AuxBet3}) & = & \mathcal{S}(\text{BetBraces}) \cup \mathcal{S}(\text{AuxBet1}) \cup \mathcal{S}(\text{AuxBet21}) \\ & = & \{\ \text{“}\}\text{”}\ \} \end{array}$$

Donc l'équation (31) est vérifiée.

$\mathcal{S}(\text{NamedConstraint}) = \{\text{“,”}, \text{“}\}\text{”}\}$

Donc les équations (41) et (42) sont vérifiées.

$$\begin{array}{rcl} \mathcal{S}(\text{AuxNamed}) & = & \mathcal{S}(\text{BetBraces}) \cup \mathcal{S}(\text{AuxBet1}) \cup \mathcal{S}(\text{AuxBet2}) \\ & & \quad \cup\, \mathcal{S}(\text{AuxBet3}) \cup \mathcal{S}(\text{AuxBet11}) \cup \mathcal{S}(\text{AuxBet21}) \\ & = & \{\ \text{“}\}\text{”}\ \} \end{array}$$

Donc l'équation (36) est vérifiée.

$$\begin{array}{rcl} \mathcal{S}(\text{NamedValue}) & = & \{\text{“,”}\} \cup \mathcal{S}(\text{AuxNamed}) \\ & = & \{\ \text{“,”}, \text{“}\}\text{”}\ \} \end{array}$$

Donc l'équation (37) est vérifiée.

$$\begin{array}{rcl} \mathcal{P}(\text{ObjIdComponent}) & = & \{\ \texttt{number}, \texttt{upper}, \texttt{lower}\ \} \\ \mathcal{S}(\text{ObjIdComponent}) & = & \mathcal{P}(\text{ObjIdComponent}) \cup \{\text{“}\}\text{”}\} \cup \mathcal{S}(\text{AuxBet1}) \\ & & \quad \cup \mathcal{S}(\text{AuxBet3}) \cup \mathcal{S}(\text{AuxBet11}) \\ & = & \{\ \texttt{number}, \texttt{upper}, \texttt{lower}, \text{“}\}\text{”}\ \} \end{array}$$

Donc l'équation (4) est vérifiée.

Il reste maintenant à vérifier le système suivant (nous avons remplacé par leur valeur les ensembles $\mathcal{P}$, sauf $\mathcal{P}$(NamedValSuf)) :

$$
\begin{array}{ll}
(8) & \{\text{“(”, “\{”, SIZE, OF}\} \cap \mathcal{S}(\text{AuxType}) = \varnothing \\
(11) & \mathcal{S}(\text{Type}) \cap \{\text{“.”}\} = \varnothing \\
(15) & \{\text{“(”}\} \cap \mathcal{S}(\text{TypeSuf}) = \varnothing \\
(17) & \{\text{“\{”, DEFINED}\} \cap \mathcal{S}(\text{BuiltInType}) = \varnothing \\
(21) & \{\text{“<”, “ :”}\} \cap \mathcal{S}(\text{Value}) = \varnothing \\
(22) & \{\text{“(”, “ :”}\} \cap \mathcal{S}(\text{AuxVal0}) = \varnothing \\
(44) & \{\text{“(”, “<”, “..”, “ :”}\} \cap \mathcal{S}(\text{SVSAux}) = \varnothing \\
(47) & \{\text{“(”}\} \cap \mathcal{S}(\text{Type}) = \varnothing
\end{array}
$$

Remarquons que $\left\{ \begin{array}{lcl} \mathcal{S}(\text{AuxType}) & \subseteq & \mathcal{S}(\text{Type}) \\ \mathcal{S}(\text{Type}) & \subseteq & \mathcal{S}(\text{AuxType}) \end{array} \right.$

Donc : $\mathcal{S}(\text{AuxType}) = \mathcal{S}(\text{Type})$

De plus : $\left\{ \begin{array}{lcl} \mathcal{S}(\text{BuiltInType}) & = & \{\text{“(”}\} \cup \mathcal{S}(\text{AuxType}) \\ \mathcal{S}(\text{TypeSuf}) & = & \mathcal{S}(\text{AuxType}) \end{array} \right.$

Nous pouvons alors regrouper les équations (8), (11), (15), (17) et (47) en une seule, et le système est équivalent à :

$$
\begin{array}{ll}
(X) & \{\text{“.”, “(”, “\{”, DEFINED, SIZE, OF}\} \cap \mathcal{S}(\text{Type}) = \varnothing \\
(21) & \{\text{“<”, “ :”}\} \cap \mathcal{S}(\text{Value}) = \varnothing \\
(22) & \{\text{“(”, “ :”}\} \cap \mathcal{S}(\text{AuxVal0}) = \varnothing \\
(44) & \{\text{“(”, “<”, “..”, “ :”}\} \cap \mathcal{S}(\text{SVSAux}) = \varnothing
\end{array}
$$

Nous avons :

$$\begin{array}{lcl}
\mathcal{S}(\text{AuxType}) & = & \mathcal{S}(\text{Type}) \cup \mathcal{S}(\text{NamedType}) \cup \{\text{“ :”}\} \\
& = & \mathcal{S}(\text{Type}) \cup \{\text{“,”, “}\}\text{”, OPTIONAL, DEFAULT, “ :”}\} \\
\mathcal{S}(\text{Type}) & = & \mathcal{S}(\text{Assignment}) \cup \{\text{“ : :=”}\} \cup \mathcal{S}(\text{AuxType}) \cup \mathcal{S}(\text{TypeSuf}) \\
& & \cup\ \mathcal{S}(\text{NamedType}) \cup \mathcal{S}(\text{ElementType}) \cup \{\text{“ :”}\} \\
& & \cup\ \mathcal{S}(\text{SubtypeValueSet})
\end{array}$$

Et il vient, en remarquant que $\mathcal{S}(\text{ElementType}) \subset \mathcal{S}(\text{NamedType}) \subset \mathcal{S}(\text{AuxType})$:

$$\begin{array}{lcl}
\mathcal{S}(\text{Type}) & = & \{\text{“ : :=”, “ :”}\} \cup \mathcal{S}(\text{Assignment}) \cup \mathcal{S}(\text{SubtypeValueSet}) \cup \mathcal{S}(\text{AuxType}) \\
& = & \{\text{“ : :=”, “ :”}\} \cup \mathcal{S}(\text{Assignment}) \cup \mathcal{S}(\text{SubtypeValueSet}) \cup \mathcal{S}(\text{Type}) \\
& & \cup\ \{\text{“,”, “}\}\text{”, OPTIONAL, DEFAULT, “ :”}\} \\
& = & \{\text{“ : :=”, “ :”, “,”, “}\}\text{”, OPTIONAL, DEFAULT}\} \cup \mathcal{S}(\text{Assignment}) \\
& & \cup\ \mathcal{S}(\text{SubtypeValueSet})
\end{array}$$

D'autre part :

$$\begin{array}{lcl}
\mathcal{S}(\text{ModuleBody}) & = & \{\ \text{END}\ \} \\
\mathcal{S}(\text{Assignment}) & = & \mathcal{S}(\text{ModuleBody}) \cup \mathcal{P}(\text{Assignment}) \\
& = & \{\ \text{END, } \texttt{upper}\text{, } \texttt{lower}\ \} \\
\mathcal{S}(\text{SubtypeValueSet}) & = & \{\ \text{“|”, “)”}\ \}
\end{array}$$

Et finalement :

$$\mathcal{S}(\text{Type}) = \{\ \text{“ : :=”, “ :”, “,”, “}\}\text{”, “|”, “)”, OPTIONAL, DEFAULT, END, } \texttt{upper}\text{, } \texttt{lower}\ \}$$

Donc l'équation (X) est satisfaite.

Remarquons que $\begin{cases} \mathcal{S}(\text{Value}) \subseteq \mathcal{S}(\text{AuxVal0}) \\ \mathcal{S}(\text{AuxVal0}) \subseteq \mathcal{S}(\text{Value}) \end{cases}$

Donc : $\mathcal{S}(\text{AuxVal0}) = \mathcal{S}(\text{Value})$

Nous pouvons subséquemment fusionner les équations (21) et (22), et le système est équivalent à :

(Y)	$\{\text{“<”, “ :”, “(”}\} \cap \mathcal{S}(\text{Value}) = \varnothing$
(44)	$\{\text{“(”, “<”, “..”, “ :”}\} \cap \mathcal{S}(\text{SVSAux}) = \varnothing$

De plus :

$$\begin{aligned}\mathcal{S}(\text{SVSAux}) &= \mathcal{S}(\text{SubtypeValueSet}) \cup \mathcal{S}(\text{SVSAux1}) \cup \mathcal{S}(\text{SVSAux2}) \\ &\quad \cup \mathcal{S}(\text{SVSAux3}) \cup \mathcal{S}(\text{SVSAux11}) \cup \mathcal{S}(\text{SVSAux21})\end{aligned}$$

Or :

$$\begin{aligned}\mathcal{S}(\text{SVSAux11}) &= \mathcal{S}(\text{SVSAux1}) \\ \mathcal{S}(\text{SVSAux1}) &= \mathcal{S}(\text{SVSAux}) \\ \mathcal{S}(\text{SVSAux2}) &= \mathcal{S}(\text{SVSAux}) \\ \mathcal{S}(\text{SVSAux3}) &= \mathcal{S}(\text{SVSAux}) \\ \mathcal{S}(\text{SVSAux21}) &= \mathcal{S}(\text{SVSAux2})\end{aligned}$$

Donc $\mathcal{S}(\text{SVSAux}) = \{\text{“|”}, \text{“)”}\}$ *et par conséquent l'équation (44) est vérifiée.*

Maintenant, il reste à calculer :

$$\begin{aligned}\mathcal{S}(\text{Value}) &= \mathcal{S}(\text{Assignment}) \cup \mathcal{S}(\text{ElementTypeSuf}) \cup \mathcal{S}(\text{AuxVal0}) \\ &\quad \cup \mathcal{S}(\text{AuxVal2}) \\ &\quad \cup \mathcal{S}(\text{SpecVal}) \cup \mathcal{S}(\text{NamedValSuf}) \cup \mathcal{S}(\text{UpperEndValue}) \\ &= \{\text{END, upper, lower}\} \cup \mathcal{S}(\text{ElementTypeSuf}) \cup \mathcal{S}(\text{AuxVal2}) \\ &\quad \cup \mathcal{S}(\text{SpecVal}) \\ &\quad \cup \mathcal{S}(\text{NamedValSuf}) \cup \mathcal{S}(\text{UpperEndValue})\end{aligned}$$

Or :

$$\begin{aligned}\mathcal{S}(\text{ElementTypeSuf}) &= \mathcal{S}(\text{ElementType}) \\ \mathcal{S}(\text{NamedValSuf}) &= \mathcal{S}(\text{NamedValue})\end{aligned}$$

D'où

$$\begin{aligned}\mathcal{S}(\text{Value}) &= \{\text{END, upper, lower, “,”, “}\}\text{”}\} \cup \mathcal{S}(\text{AuxVal2}) \cup \mathcal{S}(\text{SpecVal}) \\ &\quad \cup \mathcal{S}(\text{UpperEndValue})\end{aligned}$$

De plus :

$$\begin{aligned}\mathcal{P}(\text{AuxNamed}) &= \{\text{ “,” }\} \\ \mathcal{S}(\text{AuxVal2}) &= \mathcal{S}(\text{Value}) \cup \mathcal{P}(\text{AuxNamed}) \cup \mathcal{S}(\text{AuxBet1}) \\ &\quad \cup \mathcal{S}(\text{AuxBet11}) \cup \mathcal{S}(\text{NamedValSuf}) \\ &= \mathcal{S}(\text{Value}) \cup \{\text{“,”}, \text{“}\}\text{”}\}\end{aligned}$$

Il vient :

$$\mathcal{S}(\text{Value}) = \{\text{END, upper, lower, ",", "}\}"\} \cup \mathcal{S}(\text{SpecVal}) \cup \mathcal{S}(\text{UpperEndValue})$$

D'autre part :

$$\begin{aligned}\mathcal{S}(\text{SpecVal}) &= \mathcal{S}(\text{AuxVal0}) \cup \mathcal{S}(\text{AuxVal1}) \cup \mathcal{S}(\text{AuxVal11}) \cup \mathcal{P}(\text{AuxNamed}) \cup \mathcal{S}(\text{AuxBet2}) \cup \mathcal{S}(\text{AuxBet21}) \\ &= \mathcal{S}(\text{Value}) \cup \mathcal{S}(\text{AuxVal1}) \cup \mathcal{S}(\text{AuxVal11}) \cup \{",", "\}"\}\end{aligned}$$

Or

$$\begin{aligned}\mathcal{S}(\text{AuxVal11}) &= \mathcal{S}(\text{AuxVal1}) \\ \mathcal{S}(\text{AuxVal1}) &= \mathcal{S}(\text{Value}) \cup \mathcal{S}(\text{NamedValue}) \\ &= \mathcal{S}(\text{Value}) \cup \{",", "\}"\}\end{aligned}$$

D'où

$$\mathcal{S}(\text{SpecVal}) = \mathcal{S}(\text{Value}) \cup \{",", "\}"\}$$

Par conséquent :

$$\mathcal{S}(\text{Value}) = \{\text{END, upper, lower, ",", "}\}"\} \cup \mathcal{S}(\text{UpperEndValue})$$

Et enfin :

$$\begin{aligned}\mathcal{S}(\text{SubValSetSuf}) &= \mathcal{S}(\text{SubtypeValueSet}) \cup \mathcal{S}(\text{SVSAux}) \cup \mathcal{S}(\text{SVSAux3}) \cup \mathcal{S}(\text{SVSAux11}) \\ &= \{ \text{"|", ")" } \} \\ \mathcal{S}(\text{UpperEndValue}) &= \mathcal{S}(\text{SubValSetSuf}) \cup \mathcal{S}(\text{SVSAux2}) \cup \mathcal{S}(\text{SVSAux21}) \\ &= \mathcal{S}(\text{SubValSetSuf}) \cup \{\text{"|", ")"}\} \\ &= \{ \text{"|", ")" } \}\end{aligned}$$

D'où

$$\mathcal{S}(\text{Value}) = \{ \text{END, upper, lower, ",", "}\}", \text{"|", ")" } \}$$

Donc l'équation (Y) est vérifiée.

Conclusion Le système d'équations est entièrement vérifié, c'est-à-dire que la nouvelle grammaire ASN.1 est LL(1).

10.4 Macros

ASN.1 inclut une construction appelée « macro » qui permet au concepteur de protocoles d'étendre la syntaxe d'ASN.1. Cela signifie que l'analyseur syntaxique doit être capable de reconnaître de nouvelles notations pour les types et les valeurs, ou, en d'autres termes, il doit pouvoir étendre dynamiquement le langage qu'il reconnaît. Typiquement, cela rend ASN.1 contextuel, c'est-à-dire que l'analyse syntaxique ne peut être menée à bien qu'avec la connaissance d'un contexte (ce qui a été défini ailleurs dans la spécification). Une définition de macro (où la syntaxe pour les nouvelles valeurs et les nouveaux types sont spécifiées) est fondamentalement une extension de grammaire sans contrainte, par conséquent le spécifieur peut écrire des extensions ambiguës — rappelons qu'il a été prouvé qu'aucun algorithme en pouvait s'assurer qu'une grammaire donnée est ambiguë ou non. En plus la sémantique donnée dans ITU (1990) est obscure et incomplète.

L'idée pour traiter les macros est d'utiliser la pleine fonctionnalité de OCaml, c'est-à-dire que le résultat de l'analyse syntaxique d'une définition de macro est une paire d'analyseurs syntaxiques (des fonctions) : un pour les nouvelles notations de types et un pour les nouvelles notations de valeurs. Ils seront utilisés pour analyser leurs nouvelles notations correspondantes. En d'autres termes, nous calculons des fonctions à l'exécution qui sont elles-mêmes des analyseurs syntaxiques dédiés à la reconnaissance des nouvelles notations. Donc, dans la mesure où le résultat de l'analyse syntaxique d'une définition d'une définition de macro est un analyseur syntaxique, notre analyseur syntaxique devra effectuer certaines vérifications... sémantiques. Nous définissons ici un sous-ensemble de ce que les macros semblent devoir être, mais qui a l'avantage d'être intelligible, analysable automatiquement et aussi pas trop limité de façon à ce que l'usager ne se sente pas bridé. Pour une courte introduction et critique des macros, référez-vous à Steedman (1990). Nous utilisons le même vocabulaire pour les macros que pour les grammaires BNF que nous avons présenté à la section 10.1.1 page 233, mais avec le préfixe « macro- ». Par exemple, une *macro-règle* est une règle de grammaire définie dans une *macro-définition*.

Il y a toujours deux macro-règles spéciales `TYPE NOTATION` et `VALUE NOTATION` qui sont les points d'entrée de la macro-définition : la première définit la nouvelle syntaxe des types, et la dernière celle des valeurs. Une nouvelle notation de type (c'est-à-dire un type dénoté à l'aide d'une syntaxe définie dans une macro) et une notation de valeur (c'est-à-dire une valeur dénotée à l'aide d'une syntaxe définie dans une macro) sont respectivement appelées dans cette thèse *instance de type* et *instance de*

valeur.

Attention ! Dans (ITU, 1990), le terme « production » est utilisé dans le sens de « règle » que nous avons défini à la section 10.1.1 page 233, donc attention en lisant ce qui suit.

10.4.1 Contraintes de réalisation

Intégration incrémentale

Nous voulons intégrer le traitement des macros dans l'analyseur syntaxique de base d'une façon incrémentale. Par exemple nous voulons conserver le cœur de notre arbre de syntaxe abstraite comme un sous-ensemble strict de notre nouvel arbre étendu pour les macros. Un autre point est que nous voulons conserver un analyseur lexical inchangé. Cela implique que, bien que le macro-terminal « `"->"` » soit licite dans une macro-définition, et accepté, l'analyseur syntaxique ne reconnaîtra pas la correspondante instance. En effet, l'analyseur lexical génère en fait *deux* lexèmes : « - » et « > », car « - » n'est pas un lexème d'ASN.1. Le spécifieur aurait dû écrire « `"-" ">"` ». Pour relacher cette contrainte, nous avons préféré faire en sorte que l'analyseur lexical accepte toute sorte de symboles qui potentiellement peuvent apparaître lors d'une définition de macro (tel que « > », «! », etc.) — l'idéal serait un analyseur lexical qui soit lui aussi extensible dynamiquement.

À cause de notre intégration incrémentale, nous devons abandonner notre convention lexicale pour les identificateurs de valeurs locales dans les macro-définitions. En effet, pour des raisons historiques, elle suit la même convention que pour les identificateurs de types (ITU, 1990, § A.2.8), et rend par conséquent impossible la réutilisation de notre analyseur syntaxique de base pour la reconnaissance des valeurs. Donc nous imposons à l'intérieur des macro-définitions la même convention lexicale qu'à l'extérieur. En fait, nous identifions le lexème `local-valuereference` avec `valuereference`, et non plus avec `typereference` (Cf. 10.4.2 page 273). Notons aussi que le lexème `macroreference` est un cas particulier de `typereference` : tout ses caractères doivent être des majuscules. C'est pourquoi nous ne pouvons détecter les identificateurs des macros à l'analyse lexicale, et nous laissons cette vérification à l'analyseur sémantique, en identifiant temporairement `macroreference` et `typereference`.

Macro-lexèmes

Nous avon décidé de supprimer les macro-lexèmes `astring` et `"string"`. (Un macro-lexème est un terminal apparaissant dans la règle Symbol-

Defn et dénotant un lexème dans une macro-règle.) Ces deux macro-lexèmes sont définis respectivement dans (ITU, 1990, § A.2.7), (ITU, 1990, § A.3.10) et (ITU, 1990, § A.3.12), mais leur définition est problématique. Pour `astring`, la difficulté est la même que celle montrée à la section 10.4.1 page ci-contre. Pour `"string"`, l'exemple suivant met en évidence le problème :

```
MY-MODULE DEFINITIONS ::= BEGIN

PB MACRO ::= BEGIN
  TYPE  NOTATION ::= string
  VALUE NOTATION ::= value (VALUE BOOLEAN)
END

T     ::= TEST this is a string
val T ::= TRUE

END
```

L'analyseur syntaxique ne peut déterminer quand il doit s'arrêter de consommer des lexèmes (ici, les caractères « this », « is », « a », « string ») et peut par conséquent manger le début d'une possible déclaration de valeur qui suit (ici `val`). Le pire cas serait quand `T` est remplacé par un type selection arbitrairement long. Par souci de simplicité, nous proposons alors de supprimer ce macro-lexème `astring` et de modifier la sémantique de `"string"` : on devra maintenant mettre entre guillemets la chaîne dénotée. Dans notre exemple précédent, nous devons écrire :

```
T     ::= TEST "this is a string"
val T ::= TRUE
```

Analyse syntaxique mono-passe

Nous voulons conserver un analyseur syntaxique mono-passe et cela est possible grâce aux analyseurs syntaxiques d'ordre supérieurs de OCaml. Comme dit auparavant, le résultat d'une analyse syntaxique de macro-définition est une paire d'analyseurs dédiés, l'un aux instance de type et l'autre aux instances de valeurs. Ils sont stockés dans deux différentes tables globales, et quand l'analyseur syntaxique veut reconnaître une définition de type ou de valeur, il essaie alors d'abord de reconnaître une instance d'une macro en appelant les analyseurs disponibles dans ces tables. Si ceux-ci échouent tous, il essaie alors d'analyser comme si c'était de la syntaxe de base (sans macros). Notons que notre algorithme impose que les macro-définitions apparaissent *avant* les instances.

Les macros-définitiosn peuvent utiliser des instances d'une autre macro. Nous conservons cette possibilité,mais nous devons interdire les définitions mutuellement récursives car nous voulons conserver une seule

passe. Concrètement nous ne vérifions pas cette sorte de dépendances : l'analyse syntaxique de ces macro-définitions échouera simplement.

Les flux pour l'analyse syntaxique des instances

Nous voulons user des flux de OCaml pour l'analyse syntaxique des instances de macros. Pour relacher cette contrainte, nous ajoutons la possibilité de rebroussement limité, c'est-à-dire que si une erreur de syntaxe se produit dans un membre droit, alors l'analyseur essaie le membre droit suivant au lieu d'abandonner l'analyse (Mauny et de Rauglaudre, 1992). Si nous résumons les contraintes dont il faut tenir compte lors de la rédaction d'une macro-définition, nous avons :

1. Le spécifieur, comme toujours, doit s'assurer lui-même que sa nouvelle notation n'est pas ambiguë (car cette vérification est en général indécidable). Si le spécifieur veut une définition ambiguë, il doit se souvenir de l'ordre d'évaluation du filtre de flux correspondant (gauche-droite, haut-bas).
2. Le spécifieur doit s'assurer qu'il n'y a pas de règle récursive à gauche, qui pourrait faire boucler l'analyseur. Ceci est inhérent à notre méthode d'analyse (descendante avec rebroussement limité). Cet aspect pourrait être détecté et résolu automatiquement (avec une transformation d'Arden). Pour le moment, il faut que :
$$\forall \mathrm{A} \in \mathcal{N}, \neg(\mathrm{A} \overset{*}{\Longrightarrow} \mathrm{A}\alpha)$$
3. Le spécifieur doit s'assurer qu'il n'y pas de production inutile, c'est-à-dire de production qui génère le mot vide ε et qui n'est pas la dernière de la macro-règle. Ceci pourrait être automatiquement détecté (le problème consistant à savoir si ε appartient à un langage donné par une grammaire non contextuelle est décidable), et résolu par certaines transformations des macro-règles. Pour l'instant, il faut :
$$\mathrm{A} \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n \quad \models \quad \forall i \in [1, n], \{\varepsilon\} \subseteq \mathcal{P}(\alpha_i) \Longrightarrow i = n$$
Par exemple, considérons la macro BIND du protocole ROSE (ISO, 1989). Le macro-mot vide `empty` apparaît toujours dans la première macro-production. Ainsi :
```
Argument ::= empty | "ARGUMENT" Name type(Argument-type)
```
De cette façon l'analyseur syntaxique généré reconnaîtra toujours ε (`empty`) et n'essaiera jamais de lire la seconde production. Par conséquent, il faut écrirre plutôt :
```
Argument ::= "ARGUMENT" Name type(Argument-type) | empty
```

Détection des erreurs de syntaxe

La méthode d'analyse adoptée au point 10.4.1 de la section 10.4.1 page 268 a une répercussion importante sur la détection des erreurs de syntaxe dans les définitions de macros. Pour intégrer la reconnaissance des instances de macros à l'analyseur de base, nous ajouterons un motif en tête des filtres des fonctions d'analyse reconnaissant les types et les valeurs ASN.1. En effet, si nous le placions en dernier, l'analyseur essaierait d'abord de reconnaître un type ou une valeur de base en lieu et place d'une instance, et très probablement échouerait — sans avoir tenté de lire une instance de macro. Si l'analyseur échoue en tentant de lire une instance de macro, il croira que c'est peut-être une valeur ou un type de base qui est là, et il essaiera les motifs suivants. Si c'était bien une instance de macro que le spécifieur avait (mal) écrite, il obtiendra par conséquent une notification erreur comme si celle-ci s'était produite pour une valeur ou un type de base. C'est ici l'inconvénient du rebroussement : nous perdons *a priori* la propriété du plus long préfixe valide. Il reste cependant toujours possible de mettre au point une gestion d'erreurs syntaxiques plus précise, mais elle serait bien plus compliquée.

Importation de macros

Étant donné que OCaml ne dispose d'aucun moyen direct pour exporter des valeurs fonctionnelles et que le résultat de l'analyse d'une définition de macro est une paire de fonctions d'analyse (une pour les instances de type et une pour celles de valeur), nous préférons ignorer les importations de macros. Pour réaliser une telle importation, il faudrait imaginer un codage des fonctions OCaml vers un format symbolique, et l'opération réciproque de décodage. Cela poserait néanmoins des problèmes d'intégration à cause du typage. Néanmoins il existe une façon de simuler cette importation en allant analyser dans le module où elle est définie la macro importée, puis de poursuivre. La difficulté serait moindre car un tel analyseur est déjà disponible (c'est lui-même !) et son adaptation peu coûteuse. L'inconvénient étant que ce n'est pas une importation *stricto sensu*, donc il faut analyser la définition de macro à chaque fois qu'elle est importée.

10.4.2 Transformations de la grammaire des macros

Étape 0

Nous donnons d'abord la forme normalisée, en introduisant la structuration en sections et sous-sections, et en ayant supprimé la produc-

tion `astring` de la règle 'SymbolDefn' (cf. 10.4.1 page 268, point 10.4.1).
D'autre part, la note 2 de (ITU, 1990, § A.3.19) nous révèle que `localvaluereference` dans la règle 'LocalValueassignement' peut être VALUE. Or nous avions décidé implicitement depuis le début que nous analysions ASN.1 avec des mots-clefs réservés – dont VALUE fait partie. Par conséquent nous avons dû modifier légèrement la règle 'LocalValueassignment' pour faire apparaître explicitement le mot-clef VALUE.

MacroDefinition	→	`macroreference` MACRO " : :=" MacroSubstance
MacroSubstance	→	BEGIN MacroBody END
	|	`macroreference`
	|	Externalmacroreference
MacroBody	→	TypeProduction ValueProduction SupportingProductions
Externalmacroreference	→	`modulereference` "." `macroreference`

TypeProduction	→	TYPE NOTATION " : :=" MacroAlternativeList
ValueProduction	→	VALUE NOTATION " : :=" MacroAlternativeList

---

SupportingProductions	→	ProductionList
	|	ε
ProductionList	→	Production
	|	ProductionList Production
Production	→	`productionreference` " : :=" MacroAlternativeList
MacroAlternativeList	→	MacroAlternative
	|	MacroAlternativeList "|" MacroAlternative

MacroAlternative	→	SymbolList
SymbolList	→	SymbolElement
	|	SymbolList SymbolElement
SymbolElement	→	SymbolDefn
	|	EmbeddedDefinitions
SymbolDefn	→	`productionreference`
	|	"string"
	|	"identifier"
	|	"number"
	|	"empty"

	|	"type"
	|	"type" "(" `localtypereference` ")"
	|	"value" "(" MacroType ")"
	|	"value" "(" `localvaluereference` MacroType ")"
	|	"value" "(" VALUE MacroType ")"

EmbeddedDefinitions	→	"<" EmbeddedDefinitionList ">"
EmbeddedDefinitionList	→	EmbeddedDefinition
	|	EmbeddedDefinitionList EmbeddedDefinition
EmbeddedDefinition	→	LocalTypeassignment
	|	LocalValueassignment
LocalTypeassignment	→	`localtypereference` " : :=" MacroType
LocalValueassignment	→	`localvaluereference` MacroType " : :=" MacroValue
	|	VALUE MacroType " : :=" MacroValue

MacroType	→	`localtypereference`
	|	Type
MacroValue	→	`localvaluereference`
	|	Value

Étape 1

Nous tiendrons compte des ambiguïtés lexicales qui nous amènent à confondre `macroreference`, `productionreference`, `localtypereference`, avec `typereference` ; et `localvaluereference` avec `valuereference`. Quand le contexte le permettra, nous préciserons si le terminal `upper` dénote un identificateur de macros ou un identificateur de production en l'indiçant (respectivement $\texttt{upper}_{mac}$ et $\texttt{upper}_{prod}$).

MacroDefinition	→	$\texttt{upper}_{mac}$ MACRO " : :=" MacroSubstance
MacroSubstance	→	BEGIN MacroBody END
	|	`upper` ["." $\texttt{upper}_{mac}$]
MacroBody	→	TypeProduction ValueProduction [ProductionList]

Expansion globale de ‘ExternalMacroreference’ puis factorisation préfixe de ‘MacroSubstance’.
Option de ‘SupportingProductions’ puis expansion globale.

TypeProduction → TYPE NOTATION “ : :=” MacroAlternativeList
ValueProduction → VALUE NOTATION “ : :=” MacroAlternativeList

---

ProductionList → Production$^+$
Production → $\texttt{upper}_{prod}$ “ : :=” MacroAlternativeList
MacroAlternativeList → { MacroAlternative “|” . . . }$^+$

Arden de ‘ProductionList’.
Arden de ‘MacroAlternativeList’.

MacroAlternative → SymbolElement$^+$
SymbolElement → SymbolDefn
| EmbeddedDefinitions
SymbolDefn → $\texttt{upper}_{prod}$
| “string”
| “identifier”
| “number”
| “empty”
| “type” [“(” $\texttt{upper}_{typ}$ “)”]
| “value” “(” Bind “)”
Bind → [$\texttt{lower}_{val}$] MacroType
| VALUE MacroType

Arden de ‘SymbolList’ et expansion globale.
Factorisation préfixe et bifixe de ‘SymbolDefn’ (Création de ‘bind’.).

EmbeddedDefinitions → “<” EmbeddedDefinition$^+$ “>”
EmbeddedDefinition → $\texttt{upper}_{typ}$ “ : :=” MacroType
| $\texttt{lower}_{val}$ MacroType “ : :=” MacroValue
| VALUE MacroType “ : :=” MacroValue

Arden de ‘EmbeddedDefinitionList’ puis expansion globale.
Expansion globale de ‘LocalTypeassignment’.
Expansion globale de ‘LocalValueassignment’.

MacroType → Type
MacroValue → Value

Élimination de la variante $\texttt{upper}_{typ}$ de la règle *'MacroType'* car Type $\Longrightarrow$ `upper`
Élimination de la variante $\texttt{lower}_{val}$ de la règle *'MacroValue'* car Value $\Longrightarrow$ `lower`

Étape 2

MacroDefinition → $\texttt{upper}_{mac}$ MACRO " : :=" MacroSubstance
MacroSubstance → BEGIN MacroBody END
| `upper` ["." $\texttt{upper}_{mac}$]
MacroBody → TypeProduction ValueProduction Production*

Expansion globale de 'ProductionList'.

TypeProduction → TYPE NOTATION " : :=" { MacroAlternative "|" ... }$^+$
ValueProduction → VALUE NOTATION " : :=" { MacroAlternative "|" ... }$^+$
Production → $\texttt{upper}_{prod}$ " : :=" { MacroAlternative "|" ... }$^+$

Expansion globale de 'MacroAlternativeList'.

MacroAlternative → SymbolElement$^+$
SymbolElement → $\texttt{upper}_{prod}$
| SymbolDefn
| "<" EmbeddedDefinition$^+$ ">"
SymbolDefn → "string"
| "identifier"
| "number"
| "empty"
| "type" ["(" $\texttt{upper}_{typ}$ ")"]
| "value" "(" Bind ")"
Bind → [$\texttt{lower}_{val}$] Type

| VALUE Type

Expansion partielle de la variante $\texttt{upper}_{prod}$ de la règle 'SymbolDefn' dans la règle 'SymbolElement'.
Expansion globale de 'MacroType'.
Expansion globale de 'EmbeddedDefinitions'.

EmbeddedDefinition → $\texttt{upper}_{typ}$ " : :=" Type
| $\texttt{lower}_{val}$ Type " : :=" Value
| VALUE Type " : :=" Value

Expansion globale de 'MacroType' et 'MacroValue'.

Étape 3

MacroDefinition → $\texttt{upper}_{mac}$ MACRO " : :=" MacroSubstance
MacroSubstance → BEGIN MacroBody END
| upper ["." $\texttt{upper}_{mac}$]
MacroBody → TypeProduction VALUE NOTATION " : :="
{ MacroAlternative "|" ... }$^{+}$ Production*

Expansion globale de 'ValueProduction'.

TypeProduction → TYPE NOTATION " : :=" { MacroAlternative "|" ... }$^{+}$
Production → $\texttt{upper}_{prod}$ " : :=" { MacroAlternative "|" ... }$^{+}$

Expansion globale de 'ValueProduction'.

MacroAlternative → SymbolElement^{+}
SymbolElement → $\texttt{upper}_{prod}$
| PartElem
PartElem → SymbolDefn
| "<" EmbeddedDefinition^{+} ">"
SymbolDefn → "string"

	|	“identifier”
	|	“number”
	|	“empty”
	|	“type” [“(” $\texttt{upper}_{typ}$ “)”]
	|	“value” “(” Bind “)”
Bind	→	NamedType
	|	VALUE Type

Réduction dans la règle ‘SymbolElement’ (Création de ‘PartElem’.)
Nous savons que NamedType $\Longrightarrow$ [$\texttt{lower}_{id}$] Type
cf. (10.5.2). Donc, en réécrivant NamedType $\Longrightarrow$ [lower] Type
nous pouvons effectuer une expansion totale inverse dans la règle ‘Bind’,
et y faire apparaître une occurrence de ‘NamedType’.

EmbeddedDefinition	→	$\texttt{upper}_{typ}$ “ : :=” Type
	|	$\texttt{lower}_{val}$ Type “ : :=” Value
	|	VALUE Type “ : :=” Value

Étape 4

Le lecteur attentif aura remarqué le problème suivant :

$$\mathcal{P}(\text{MacroAlternative}) \cap \mathcal{P}(\text{Production}) = \{\texttt{upper}\}$$

empêchant donc la règle ‘MacroBody’ d’être LL(1). Pour éliminer cette difficulté, nous allons transformer cette règle et c’est l’objet de cette section que d’en présenter les métamorphoses.

MacroBody	→	TypeProduction VALUE NOTATION “ : :=” MacroSuf
MacroSuf	→	{ MacroAlternative “|” ... }$^+$ Production*

Expansion totale inverse. (Création de la règle ‘MacroSuf’.)

MacroSuf	→	MacroAlternative [“|” { MacroAlternative “|” ... }$^+$] Production*

MacroSuf → SymbolElement$^+$ [“|” { MacroAlternative “|” . . . }$^+$] Production*

Expansion totale de ‘MacroAlternative’.

MacroSuf → SymbolElement Cont
Cont → SymbolElement* [“|” { MacroAlternative “|” . . . }$^+$] Production*

MacroSuf → SymbolElement [Cont]
Cont → SymbolElement$^+$ [“|” { MacroAlternative “|” . . . }$^+$] Production*
| “|” { MacroAlternative “|” . . . }$^+$ Production*
| Production$^+$

Option de ’Cont’.

MacroSuf → SymbolElement [Cont]
Cont → SymbolElement [Cont]
| “|” MacroSuf
| Production$^+$

Nous reconnaissons ‘Cont’ et ‘MacroSuf’. (Expansions totales inverses.)

MacroSuf → SymbolElement [Cont]
Cont → `upper` [Cont]
| PartElem [Cont]
| “|” MacroSuf
| Production Production*

Expansion totale de ‘SymbolElement’ dans ‘Cont’.

MacroSuf → SymbolElement [Cont]
Cont → PartElem [Cont]
| “|” MacroSuf
| `upper` [Cont]
| `upper`$_{prod}$ “ : :=” { MacroAlternative “|” . . . }$^+$ Production*

Expansion totale de ‘Production’ dans ‘Cont’.

MacroSuf	→	SymbolElement [Cont]
Cont	→	PartElem [Cont]
	|	“|” MacroSuf
	|	`upper` [ContSuf]
ContSuf	→	Cont
	|	“ : :=” MacroSuf

Factorisation préfixe de ‘Cont’. (Création de la règle ‘ContSuf’.)

Bilan

MacroDefinition	→	$\texttt{upper}_{mac}$ MACRO “ : :=” MacroSubstance
MacroSubstance	→	BEGIN MacroBody END
	|	`upper` [“.” $\texttt{upper}_{mac}$]
MacroBody	→	TypeProduction VALUE NOTATION “ : :=” MacroSuf

TypeProduction	→	TYPE NOTATION “ : :=” { MacroAlternative “|” . . . }$^+$
MacroAlternative	→	SymbolElement$^+$

MacroSuf	→	SymbolElement [Cont]
Cont	→	PartElem [Cont]
	|	“|” MacroSuf
	|	`upper` [ContSuf]
ContSuf	→	Cont
	|	“ : :=” MacroSuf

SymbolElement	→	$\texttt{upper}_{prod}$
	|	PartElem
PartElem	→	SymbolDefn

	|	“<” EmbeddedDefinition$^+$ “>”
SymbolDefn	→	“string”
	|	“identifier”
	|	“number”
	|	“empty”
	|	“type” [“(” $\texttt{upper}_{typ}$ “)”]
	|	“value” “(” Bind “)”
Bind	→	NamedType
	|	VALUE Type

EmbeddedDefinition	→	$\texttt{upper}_{typ}$ “ : :=” Type
	|	$\texttt{lower}_{val}$ Type “ : :=” Value
	|	VALUE Type “ : :=” Value

10.4.3 Nouvelle grammaire complète d’ASN.1

Nous devons greffer cette nouvelle grammaire des macros sur celle de base, le document ITU (1990) ne disant pas explicitement comment s’effectue cette greffe. En fait il faut intégrer la définition des macros parmi les définitions de types et de valeurs de base. À partir de la forme initiale, nous obtenons les transformations suivantes :

Assignment	→	$\texttt{upper}_{typ}$ “ : :=” Type
	|	$\texttt{lower}_{val}$ Type “ : :=” Value

Assignment	→	$\texttt{upper}_{typ}$ “ : :=” Type
	|	$\texttt{lower}_{val}$ Type “ : :=” Value
	|	MacroDefinition

Assignment	→	$\texttt{upper}_{typ}$ “ : :=” Type
	|	$\texttt{lower}_{val}$ Type “ : :=” Value
	|	$\texttt{upper}_{mac}$ MACRO “ : :=” MacroSubstance

Expansion globale de 'MacroDefinition'.

Assignment	→	`upper` AssSuf
	|	`lower`$_{val}$ Type " : :=" Value
AssSuf	→	MACRO " : :=" MacroSubstance
	|	" : :=" Type

Factorisation préfixe de 'MacroDefinition'.

Donc finalement, la forme finale de la nouvelle grammaire complète de *full*-ASN.1 est :

MODULES

ModuleDefinition	→	ModuleIdentifier DEFINITIONS [TagDefault TAGS] " : :=" BEGIN [ModuleBody] END
ModuleIdentifier	→	`upper`$_{mod}$ ["{" ObjIdComponent$^+$ "}"]
ObjIdComponent	→	`number`
	|	`upper`$_{mod}$ "." `lower`$_{val}$
	|	`lower` ["(" ClassNumber ")"]
TagDefault	→	EXPLICIT
	|	IMPLICIT
ModuleBody	→	[Exports] [Imports] Assignment$^+$

Exports	→	EXPORTS {Symbol “,” . . . }* “ ;”
Imports	→	IMPORTS SymbolsFromModule* “ ;”
SymbolsFromModule	→	{Symbol “,” . . . }$^+$ FROM ModuleIdentifier
Symbol	→	`upper`$_{typ}$
	|	`lower`$_{val}$

Assignment	→	`upper` AssSuf
	|	`lower`$_{val}$ Type “ : :=” Value
AssSuf	→	MACRO “ : :=” MacroSubstance
	|	“ : :=” Type

TYPES

Type	→	`lower`$_{id}$ “<” Type
	|	`upper` [“.” `upper`$_{typ}$] SubtypeSpec*
	|	NULL SubtypeSpec*
	|	AuxType
AuxType	→	“[” [Class] ClassNumber “]” [TagDefault] Type
	|	BuiltInType SubtypeSpec*
	|	SetSeq [TypeSuf]
SetSeq	→	SET
	|	SEQUENCE
TypeSuf	→	SubtypeSpec$^+$
	|	“{” {ElementType “,” . . . }* “}” SubtypeSpec*
	|	[SIZE SubtypeSpec] OF Type

BuiltInType	→	BOOLEAN
	|	INTEGER [“{” {NamedNumber “,” . . . }$^+$ “}”]
	|	BIT STRING [“{” {NamedBit “,” . . . }$^+$ “}”]
	|	OCTET STRING
	|	CHOICE “{” {NamedType “,” . . . }$^+$ “}”
	|	ANY [DEFINED BY `lower`$_{id}$]
	|	OBJECT IDENTIFIER
	|	ENUMERATED “{” {NamedNumber “,” . . . }$^+$ “}”
	|	REAL
	|	“NumericString”

		“PrintableString”
		“TeletexString”
		“T61String”
		“VideotexString”
		“VisibleString”
		“ISO646String”
		“IA5String”
		“GraphicString”
		“GeneralString”
		EXTERNAL
		“UTCTime”
		“GeneralizedTime”
		“ObjectDescriptor”

NamedType	→	`lower` [“<”] Type
			`upper` [“.” `upper`$_{typ}$] SubtypeSpec*
			NULL SubtypeSpec*
			AuxType

NamedNumber	→	`lower`$_{id}$ “(” AuxNamedNum “)”
AuxNamedNum	→	[“-”] `number`
			[`upper`$_{mod}$ “.”] `lower`$_{val}$

NamedBit	→	`lower`$_{id}$ “(” ClassNumber “)”

ElementType	→	NamedType [ElementTypeSuf]
			COMPONENTS OF Type
ElementTypeSuf	→	OPTIONAL
			DEFAULT Value

Class	→	UNIVERSAL
			APPLICATION
			PRIVATE
ClassNumber	→	`number`
			[`upper`$_{mod}$ “.”] `lower`$_{val}$

VALEURS

Value	→	AuxVal0
	|	`upper` AuxVal1
	|	`lower` [AuxVal2]
	|	["-"] `number`
AuxVal0	→	BuiltInValue
	|	AuxType " :" Value
	|	NULL [SpecVal]
AuxVal1	→	SpecVal
	|	"." AuxVal11
AuxVal2	→	["<" Type] " :" Value
AuxVal11	→	$\texttt{upper}_{typ}$ SpecVal
	|	$\texttt{lower}_{val}$
SpecVal	→	SubtypeSpec* " :" Value

BuiltInValue	→	TRUE
	|	FALSE
	|	PLUS-INFINITY
	|	MINUS-INFINITY
	|	`basednum`
	|	`string`
	|	"{" [BetBraces] "}"

BetBraces	→	AuxVal0 [AuxNamed]
	|	"-" `number` [AuxNamed]
	|	`lower` [AuxBet1]
	|	`upper` AuxBet2
	|	`number` [AuxBet3]
AuxBet1	→	"(" ClassNumber ")" ObjIdComponent*
	|	AuxNamed
	|	AuxVal2 [AuxNamed]
	|	"-" `number` [AuxNamed]
	|	AuxVal0 [AuxNamed]
	|	`lower` [AuxBet11]
	|	`number` [AuxBet3]
	|	`upper` AuxBet2

AuxBet2	→	SpecVal [AuxNamed]
	|	"." AuxBet21
AuxBet3	→	ObjIdComponent$^+$
	|	AuxNamed
AuxBet11	→	"(" ClassNumber ")" ObjIdComponent*
	|	ObjIdComponent$^+$
	|	AuxVal2 [AuxNamed]
	|	AuxNamed
AuxBet21	→	`upper`$_{typ}$ SpecVal [AuxNamed]
	|	`lower`$_{val}$ [AuxBet3]
AuxNamed	→	"," {NamedValue "," ...}$^+$
NamedValue	→	`lower` [NamedValSuf]
	|	`upper` AuxVal1
	|	["-"] `number`
	|	AuxVal0
NamedValSuf	→	Value
	|	AuxVal2

SOUS-TYPES

SubtypeSpec	→	"(" {SubtypeValueSet "|" ...}$^+$ ")"
SubtypeValueSet	→	INCLUDES Type
	|	MIN SubValSetSuf
	|	FROM SubtypeSpec
	|	SIZE SubtypeSpec
	|	WITH InnerTypeSuf
	|	SVSAux

SubValSetSuf	→	["<"] ".." ["<"] UpperEndValue
UpperEndValue	→	Value
	|	MAX

InnerTypeSuf	→	COMPONENT SubtypeSpec
	|	COMPONENTS MultipleTypeConstraints
MultipleTypeConstraints	→	"{" ["..." ","] {[NamedConstraint] "," ...} "}"
NamedConstraint	→	`lower`$_{id}$ [SubtypeSpec] [PresenceConstraint]
	|	SubtypeSpec [PresenceConstraint]
	|	PresenceConstraint
PresenceConstraint	→	PRESENT

	|	ABSENT
	|	OPTIONAL

SVSAux	→	BuiltInValue [SubValSetSuf]
	|	AuxType " :" SVSAux
	|	NULL [SVSAux3]
	|	`upper` SVSAux1
	|	`lower` [SVSAux2]
	|	["-"] `number` [SubValSetSuf]
SVSAux1	→	SubtypeSpec* " :" SVSAux
	|	"." SVSAux11
SVSAux2	→	" :" SVSAux
	|	".." ["<"] UpperEndValue
	|	"<" SVSAux21
SVSAux3	→	SubtypeSpec* " :" SVSAux
	|	SubValSetSuf
SVSAux11	→	$\texttt{upper}_{typ}$ SubtypeSpec* " :" SVSAux
	|	$\texttt{lower}_{val}$ [SubValSetSuf]
SVSAux21	→	Type " :" SVSAux
	|	".." ["<"] UpperEndValue

MACROS

MacroSubstance	→	BEGIN MacroBody END
	|	`upper` [". " $\texttt{upper}_{mac}$]
MacroBody	→	TypeProduction VALUE NOTATION " : :=" MacroSuf

TypeProduction	→	TYPE NOTATION " : :=" { MacroAlternative "|" ... }$^+$
MacroAlternative	→	SymbolElement$^+$

MacroSuf	→	SymbolElement [Cont]
Cont	→	PartElem [Cont]
	|	"|" MacroSuf
	|	`upper` [ContSuf]
ContSuf	→	Cont
	|	" : :=" MacroSuf

SymbolElement	→	$\texttt{upper}_{prod}$
	|	PartElem
PartElem	→	SymbolDefn
	|	"<" EmbeddedDefinition$^+$ ">"
SymbolDefn	→	"string"
	|	"identifier"
	|	"number"
	|	"empty"
	|	"type" ["(" $\texttt{upper}_{typ}$ ")"]
	|	"value" "(" Bind ")"
Bind	→	NamedType
	|	VALUE Type

EmbeddedDefinition	→	$\texttt{upper}_{typ}$ " : :=" Type
	|	$\texttt{lower}_{val}$ Type " : :=" Value
	|	VALUE Type " : :=" Value

10.4.4 Preuve de la propriété LL(1) de la grammaire étendue

Nous n'allons pas vérifier la propriété LL(1) de la grammaire étendue (avec les macros) *ex nihilo*, mais incrémentalement, à partir des résultats donnés en (10.3 page 246).

Équation $\mathbf{P}_1$

Il est évident de constater que la sous-grammaire des macros n'engendre pas de récursivités à gauche.

Équation $\mathbf{P}_2$

Il est aisé de vérifier que l'intersection des premiers (lexèmes) de chaque alternative est vide.

Équation $\mathbf{P}_3$

Avant de commencer, notons que la modification de la règle *'Assignment'* n'engendre pas de nouvelles contraintes. Pour ce qui est de la sous-grammaire des macros, il vient :

Règle	Contraintes
MacroSubstance	$\{\text{“.”}\} \cap \mathcal{S}(\text{MacroSubstance}) = \emptyset$
TypeProduction	$\{\text{“\|”}\} \cap \mathcal{S}(\text{TypeProduction}) = \emptyset$
MacroAlternative	$\mathcal{P}(\text{SymbolElement}) \cap \mathcal{S}(\text{MacroAlternative}) = \emptyset$
MacroSuf	$\mathcal{P}(\text{Cont}) \cap \mathcal{S}(\text{MacroSuf}) = \emptyset$
Cont	$\mathcal{P}(\text{Cont}) \cap \mathcal{S}(\text{Cont}) = \emptyset$ $\mathcal{P}(\text{ContSuf}) \cap \mathcal{S}(\text{Cont}) = \emptyset$
EmbeddedDefinition	$\mathcal{P}(\text{EmbeddedDefinition}) \cap \{\text{“>”}\} = \emptyset$
SymbolDefn	$\{\text{“(”}\} \cap \mathcal{S}(\text{SymbolDefn}) = \emptyset$

Soit :

(1) $\{\text{“.”}\} \cap \mathcal{S}(\text{MacroSubstance}) = \emptyset$
(2) $\{\text{“\|”}\} \cap \mathcal{S}(\text{TypeProduction}) = \emptyset$
(3) $\mathcal{P}(\text{SymbolElement}) \cap \mathcal{S}(\text{MacroAlternative}) = \emptyset$
(4) $\mathcal{P}(\text{Cont}) \cap \mathcal{S}(\text{MacroSuf}) = \emptyset$
(5) $\mathcal{P}(\text{Cont}) \cap \mathcal{S}(\text{Cont}) = \emptyset$
(6) $\mathcal{P}(\text{ContSuf}) \cap \mathcal{S}(\text{Cont}) = \emptyset$
(7) $\mathcal{P}(\text{EmbeddedDefinition}) \cap \{\text{“>”}\} = \emptyset$
(8) $\{\text{“(”}\} \cap \mathcal{S}(\text{SymbolDefn}) = \emptyset$

Calculons d'abord les premiers :

$\mathcal{P}(\text{SymbolElement})$ = { `upper`, “<”, “`string`”, “`identifier`”, “`number`”, “`empty`”, “`type`”, “`value`” }

$\mathcal{P}(\text{Cont})$ = { `upper`, “<”, “`string`”, “`identifier`”, “`number`”, “`empty`”, “`type`”, “`value`”, “|” }

$\mathcal{P}$(ContSuf) = { `upper`, “<”, “`string`”, “`identifier`”, “`number`”, “`empty`”, “`type`”, “`value`”, “|”, “ : :=” }

$\mathcal{P}$(EmbeddedDefinition) = { `upper`, `lower`, VALUE }

Puis les suivants :

$$
\begin{aligned}
\mathcal{S}(\text{MacroSubstance}) &= \mathcal{S}(\text{AssSuf}) \\
&= \mathcal{S}(\text{Assignment}) \\
&= \{ \text{END}, \texttt{upper}, \texttt{lower} \}
\end{aligned}
$$

$$
\mathcal{S}(\text{TypeProduction}) = \{ \text{VALUE} \}
$$

$$
\begin{aligned}
\mathcal{S}(\text{MacroAlternative}) &= \{\text{“|”}\} \cup \mathcal{S}(\text{TypeProduction}) \\
&= \{ \text{VALUE}, \text{“|”} \}
\end{aligned}
$$

$$
\mathcal{S}(\text{ContSuf}) = \mathcal{S}(\text{Cont})
$$

$$
\begin{aligned}
\mathcal{S}(\text{MacroSuf}) &= \mathcal{S}(\text{MacroBody}) \cup \mathcal{S}(\text{Cont}) \cup \mathcal{S}(\text{ContSuf}) \\
&= \{\text{END}\} \cup \mathcal{S}(\text{Cont}) \cup \mathcal{S}(\text{ContSuf}) \\
&= \{\text{END}\} \cup \mathcal{S}(\text{Cont})
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{S}(\text{Cont}) &= \mathcal{S}(\text{MacroSuf}) \cup \mathcal{S}(\text{ContSuf}) \\
&= \mathcal{S}(\text{MacroSuf}) \cup \mathcal{S}(\text{Cont}) \\
&= \mathcal{S}(\text{MacroSuf})
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{S}(\text{SymbolDefn}) &= \mathcal{S}(\text{PartElem}) \\
&= \mathcal{P}(\text{Cont}) \cup \mathcal{S}(\text{Cont}) \cup \mathcal{S}(\text{SymbolElement}) \\
&= \mathcal{P}(\text{Cont}) \cup \mathcal{S}(\text{Cont}) \cup \mathcal{S}(\text{MacroAlternative}) \cup \mathcal{S}(\text{MacroSuf}) \\
&= \mathcal{S}(\text{MacroSuf}) \cup \{\text{VALUE}, \text{“|”}, \texttt{upper}, \text{“<”}, \text{“}\texttt{string}\text{”}, \\
&\quad \text{“}\texttt{identifier}\text{”}, \text{“}\texttt{number}\text{”}, \text{“}\texttt{empty}\text{”}, \text{“}\texttt{type}\text{”}, \text{“}\texttt{value}\text{”} \}
\end{aligned}
$$

D’où :

$$\mathcal{S}(\text{MacroSuf}) = \mathcal{S}(\text{Cont}) = \mathcal{S}(\text{ContSuf}) = \{\text{END}\}$$

Et

$$\begin{aligned}\mathcal{S}(\text{SymbolDefn}) &= \{\ \text{END, VALUE, “|”, } \texttt{upper}\text{, “<”, “}\texttt{string}\text{”,}\\ &\quad \text{“}\texttt{identifier}\text{”, “}\texttt{number}\text{”, “}\texttt{empty}\text{”, “}\texttt{type}\text{”, “}\texttt{value}\text{” }\}\end{aligned}$$

Donc le système (1)-(8) est vérifié.

Il reste à s'assurer que les non-terminaux apparaissant dans la sous-grammaire des macros *et* dans la grammaire de base n'induisent pas des ensembles $\mathcal{S}$ qui invalideraient la propriété LL(1) de la grammaire de base. Les non-terminaux dans ce cas à examiner sont 'NamedType', 'Type' et 'Value'.

Nous avions : $\mathcal{S}$(NamedType) = { “,”, “}”, OPTIONAL, DEFAULT }. Maintenant :

$$\begin{aligned}\mathcal{S}(\text{NamedType}) &= \mathcal{S}(\text{Bind}) \cup \{\text{“,”, “}\}\text{”, OPTIONAL, DEFAULT}\}\\ &= \{\ \text{“)”, “,”, “}\}\text{”, OPTIONAL, DEFAULT }\}\end{aligned}$$

Considérons maintenant où est utilisé $\mathcal{S}$(NamedType). Il est utilisé d'une part pour vérifier les équations (18) et (19) données à la section 10.3.2 page 255, et d'autre part pour calculer $\mathcal{S}$(AuxType), qui sert lui-même uniquement à calculer $\mathcal{S}$(Type). Les équations (18) et (19) sont toujours vérifiées et pour ce qui est de $\mathcal{S}$(Type), celui-ci reste invariant car il contenait déjà “)”.

Les occurrences de 'Type' et 'Value' dans la sous-grammaire des macros introduisent dans $\mathcal{S}$(Type) et $\mathcal{S}$(Value) le symbole terminal “>”. Or celui-ci n'existe pas dans la grammaire de base, donc il ne peut interférer dans les calculs de (10.3.2).

Conclusion : La grammaire étendue est LL(1).

10.4.5 Extension de l'arbre de syntaxe abstraite

Nous allons expliquer pourquoi il est nécessaire d'étendre l'arbre de syntaxe abstraite de base pour pouvoir traiter les macros et comment le faire simplement et incrémentalement. Le problème se divise en deux parties disjointes : le cas des instances de type et celui des instances de valeur.

Instances de types

Le sixième paragraphe de (ITU, 1990, § A.1) nous apprend qu'une instance de type peut dépendre d'une instance de valeur. Considérons l'exemple suivant :

```
SAMPLE DEFINITIONS ::= BEGIN
```

```
TEST MACRO ::= BEGIN
  TYPE  NOTATION ::= empty
  VALUE NOTATION ::= value (VALUE BOOLEAN) | value (VALUE INTEGER)
END

T ::= TEST

END
```

Quel est donc le type de `T` ? Réponse : cela dépend. Si une instance de valeur apparaissait dans le module, alors `T` pourrait être soit booléen, soit entier. En l'absence d'une telle instance de valeur, nous devons le rejeter comme étant incorrectement défini... (Attention : `T` n'est même pas le type non spécifié NULL.) C'est là qu'arrive au galop la fin de ce sixième paragraphe qui nous apprend qu'en fait nous devons considérer une instance de type comme un type choix 'ChoiceType' (« *[...] the use of the new type notation is similar to a CHOICE [...].*). Ainsi dans notre exemple, cela nous amène à penser que `T` serait équivalent, dans un sens qui resterait à définir, à :

```
 T ::= CHOICE { field-0 BOOLEAN,
                field-1 INTEGER }
```

Toutefois, les types possibles pour une instance de macro (c'est-à-dire les champs du type choix associé) peuvent contenir des identificateurs locaux à la macro. Par exemple :

```
MACRO-INSIDE DEFINITIONS ::= BEGIN

PAIR MACRO ::= BEGIN
  TYPE NOTATION ::= "TYPEX" "=" type(LT1) "TYPEY" "=" type(LT2)

  VALUE NOTATION ::= "(" "X" "=" value(lv1 LT1) ","
                         "Y" "=" value(lv2 LT2)
                      <VALUE SEQUENCE {LT1, LT2} ::= {lv1, lv2}>
                     ")"
END

T1 ::= PAIR
         TYPEX = INTEGER
         TYPEY = BOOLEAN

END
```

Dans ce cas, il faut produire une *fermeture de type*, c'est-à-dire une paire constituée d'un type (choix) et d'un environnement (un ensemble de définitions locales à la macro qui contiendrait ici celles de `LT1` et `LT2`). Nous déciderons donc que toute instance de type sera équivalente à une telle

fermeture. Nous allons pour ce faire étendre l'arbre de syntaxe abstraite en rajoutant un nœud « fermeture de type », c'est-à-dire un constructeur supplémentaire `TClos` au type OCaml `Desc` :

```
and desc = ...
  | TClos of desc * ctxt
  | ...
and ctxt = def list
```

Instances de valeurs

La section précédente consacrée aux instances de types nous amène naturellement à concevoir les instances de valeur comme des fermetures de valeurs, c'est-à-dire une paire formée d'une valeur et d'un environnement contenant des définitions locales à la macro. Plus exactement, ce sont des valeurs choisies 'ChoiceValue' implicites, couplées à un environnement. Pour réaliser cela, nous devons ajouter un nœud supplémentaire à l'arbre de syntaxe abstraite, c'est-à-dire un constructeur `VClos` au type OCaml `Value` :

```
    and v_desc = ...
  | VClos of value * ctxt    | ...
and ctxt = def list
```

10.5 Transformations grammaticales

10.5.1 Transformations de la grammaire des modules

Étape 0

Nous présentons d'abord la partie correspondant à la spécification des modules ASN.1 Nous introduisons le découpage et les regroupements de règles.

ModuleDefinition	→	ModuleIdentifier DEFINITIONS TagDefault " : :=" BEGIN ModuleBody END

ModuleIdentifier	→	`modulereference` AssignedIdentifier
AssignedIdentifier	→	ObjectIdentifierValue
	|	ε
ObjectIdentifierValue	→	"{" ObjIdComponentList "}"
	|	"{" DefinedValue ObjIdComponentList "}"
ObjIdComponentList	→	ObjIdComponent
	|	ObjIdComponent ObjIdComponentList
ObjIdComponent	→	NameForm
	|	NumberForm
	|	NameAndNumberForm
NameForm	→	`identifier`
NumberForm	→	`number`
	|	DefinedValue
NameAndNumberForm	→	`identifier` "(" NumberForm ")"

TagDefault	→	EXPLICIT TAGS
	|	IMPLICIT TAGS
	|	ε

ModuleBody	→	Exports Imports AssignmentList
	|	ε
Exports	→	EXPORTS SymbolsExported " ;"
	|	ε
Imports	→	IMPORTS SymbolsImported " ;"
	|	ε
SymbolsExported	→	SymbolList
	|	ε
SymbolsImported	→	SymbolsFromModuleList
	|	ε
SymbolList	→	Symbol
	|	Symbol "," SymbolList
SymbolsFromModuleList	→	SymbolsFromModule
	|	SymbolsFromModuleList SymbolsFromModule
Symbol	→	`typereference`
	|	`valuereference`
SymbolsFromModule	→	SymbolList FROM ModuleIdentifier

AssignmentList	→	Assignment
	|	AssignmentList Assignment
Assignment	→	TypeAssignment
	|	ValueAssignment
TypeAssignment	→	`typereference` " : :=" Type
ValueAssignment	→	`valuereference` Type " : :=" Value

DefinedType	→	Externaltypereference
	|	`typereference`
DefinedValue	→	Externalvaluereference
	|	`valuereference`
Externaltypereference	→	`modulereference` "." `typereference`
Externalvaluereference	→	`modulereference` "." `valuereference`

Étape 1

Nous substituons les identificateurs de terminaux (lexicalement) ambigus, en conservant si possible l'information sur leur sémantique originelle (qui apparaît alors en indice). D'autre part, nous appliquons l'option jusqu'à faire disparaître tous les ε, et nous utilisons le lemme d'Arden pour introduire les opérateurs rationnels.

ModuleDefinition	→	ModuleIdentifier DEFINITIONS [TagDefault TAGS] " : :=" BEGIN [ModuleBody] END

Cf. 'TagDefault' et 'ModuleBody'.

ModuleIdentifier	→	$\texttt{upper}_{mod}$ ["{" ObjIdComponent$^+$ "}"]
ObjIdComponent	→	$\texttt{lower}_{id}$
	|	NumberForm
	|	$\texttt{lower}_{id}$ "(" NumberForm ")"

NumberForm	→	`number`
	|	DefinedValue

Option puis expansion globale de ‘AssignedIdentifier’.
Arden de ‘ObjIdComponentList’.
Expansion globale de ‘NameForm’.
Expansion globale de ‘NameAndNumberForm’.
Élimination de la seconde production de ‘ObjectIdentifierValue’ car :
ObjIdComponentList ⇒ ObjIdComponent ObjIdComponentList
⇒ NumberForm ObjIdComponentList ⇒ DefinedValue ObjIdComponentList.
Expansion globale de ‘ObjIdComponentList’.
Expansion globale de *‘ObjectIdentifierValue’*.
Attention ! Nous conservons ObjectIdentifierValue → “{” ObjIdComponent$^+$ “}” dans la section (10.5.3).

TagDefault	→	EXPLICIT
	|	IMPLICIT

Option puis expansion suffixe de ‘TagDefault’.

ModuleBody	→	[Exports] [Imports] AssignmentList
Exports	→	EXPORTS [SymbolList] “ ;”
Imports	→	IMPORTS [SymbolsFromModuleList] “ ;”
SymbolList	→	{Symbol “,” . . . }$^+$
SymbolsFromModuleList	→	SymbolsFromModule$^+$
Symbol	→	`typereference`
	|	`valuereference`
SymbolsFromModule	→	SymbolList FROM ModuleIdentifier

Option de ‘Exports’ et de ‘Imports’.
Option puis expansion globale de ‘SymbolsExported’.
Option puis expansion globale de ‘SymbolsImported’.
Arden de ‘SymbolList’ et ‘SymbolsFromModuleList’.

AssignmentList	→	Assignment$^+$
Assignment	→	`upper`$_{typ}$ “ : :=” Type
	|	`lower`$_{val}$ Type “ : :=” Value

Arden de *'AssignmentList'*.
Expansion globale de 'TypeAssignment' et 'ValueAssignment'.

DefinedType → `upper`$_{mod}$ "." `upper`$_{typ}$
| `upper`$_{typ}$

DefinedValue → `upper`$_{mod}$ "." `lower`$_{val}$
| `lower`$_{val}$

Expansion globale de 'ExternalTypeReference' et 'ExternalValueReference'.

Étape 2

ModuleDefinition → ModuleIdentifier
DEFINITIONS
[TagDefault TAGS]
" : :=="
BEGIN
[ModuleBody]
END

ModuleIdentifier → `upper`$_{mod}$ ["{" ObjIdComponent$^+$ "}"]
ObjIdComponent → `number`
| `upper`$_{mod}$ "." `lower`$_{val}$
| `lower` ["(" ClassNumber ")"]

Expansion totale de *'DefinedValue'*.
Factorisation préfixe (`lower`).
Élimination de 'NumberForm' car 'NumberForm' = 'ClassNumber' (Cf. section 10.5.2).

TagDefault → EXPLICIT
| IMPLICIT

ModuleBody → [Exports] [Imports] Assignment$^+$
Exports → EXPORTS {Symbol "," ...}* ";"
Imports → IMPORTS SymbolsFromModule* ";"

SymbolsFromModule	→	{Symbol “,” . . . }$^+$ FROM ModuleIdentifier
Symbol	→	`upper`$_{typ}$
	|	`lower`$_{val}$

Expansion globale de *‘AssignmentList’*.
Expansion globale de ‘SymbolList’.
Expansion globale de ‘SymbolsFromModuleList’.

Assignment	→	`upper`$_{typ}$ “ : :=” Type
	|	`lower`$_{val}$ Type “ : :=” Value

‘DefinedType’ est déplacée dans la section (10.5.2).
‘DefinedValue’ est déplacée dans les sections 10.5.2 et (10.5.3).

10.5.2 Transformations de la grammaire des types

Étape 0

Nous donnons d’abord ici la forme normalisée de la section consacrée à la spécification des types ASN.1, après restructuration. Notons que nous récupérons les règles *‘DefinedType’* et *‘DefinedValue’* en provenance de la section 10.5.1 page 292. Les règles *‘Subtype’* et *‘ParentType’* sont présentes dans cette section, et non pas dans la section 10.5.4 page 322, pour faciliter la lecture. *‘SubtypeSpec’* est présente dans la section (10.5.4) . Nous noterons aussi que *‘ClassNumber’* est une règle d’entrée mixte (section 10.5.1 page précédente).

Type	→	DefinedType
	|	BuiltInType
	|	Subtype

DefinedType	→	`upper` [“.” `upper`$_{typ}$]
DefinedValue	→	[`upper`$_{mod}$ “.”] `lower`$_{val}$

Factorisation préfixe de *‘DefinedType’*.
Factorisation suffixe de *‘DefinedValue’*.

BuiltInType	→	BooleanType
	|	IntegerType
	|	BitStringType
	|	OctetStringType
	|	NullType
	|	SequenceType
	|	SequenceOfType
	|	SetType
	|	SetOfType
	|	ChoiceType
	|	SelectionType
	|	TaggedType
	|	AnyType
	|	ObjectIdentifierType
	|	CharacterStringType
	|	UsefulType
	|	EnumeratedType
	|	RealType

NamedType	→	`identifier` Type
	|	Type
	|	SelectionType

Subtype	→	ParentType SubtypeSpec
	|	SET SizeConstraint OF Type
	|	SEQUENCE SizeConstraint OF Type
ParentType	→	Type
SizeConstraint	→	SIZE SubtypeSpec

IntegerType	→	INTEGER
	|	INTEGER "{" NamedNumberList "}"
NamedNumberList	→	NamedNumber
	|	NamedNumberList "," NamedNumber
NamedNumber	→	`identifier` "(" SignedNumber ")"
	|	`identifier` "(" DefinedValue ")"
SignedNumber	→	["-"] `number`

BooleanType	→	BOOLEAN
EnumeratedType	→	ENUMERATED “{” Enumeration “}”
Enumeration	→	NamedNumber
	|	Enumeration “,” NamedNumber
RealType	→	REAL
BitStringType	→	BIT STRING
	|	BIT STRING “{” NamedBitList “}”
NamedBitList	→	NamedBit
	|	NamedBitList “,” NamedBit
NamedBit	→	`identifier` “(” `number` “)”
	|	`identifier` “(” DefinedValue “)”
OctetStringType	→	OCTET STRING
SequenceOfType	→	SEQUENCE OF Type
	|	SEQUENCE
SetOfType	→	SET OF Type
	|	SET
NullType	→	NULL
SequenceType	→	SEQUENCE “{” ElementTypeList “}”
	|	SEQUENCE “{” “}”
SetType	→	SET “{” ElementTypeList “}”
	|	SET “{” “}”
ElementTypeList	→	ElementType
	|	ElementTypeList “,” ElementType
ElementType	→	NamedType
	|	NamedType OPTIONAL

	|	NamedType DEFAULT Value
	|	COMPONENTS OF Type

ChoiceType	→	CHOICE "{" AlternativeTypeList "}"
AlternativeTypeList	→	NamedType
	|	AlternativeTypeList "," NamedType

SelectionType	→	`identifier` "<" Type

TaggedType	→	Tag Type
	|	Tag IMPLICIT Type
	|	Tag EXPLICIT Type
Tag	→	"[" Class ClassNumber "]"
ClassNumber	→	`number`
	|	DefinedValue
Class	→	UNIVERSAL
	|	APPLICATION
	|	PRIVATE
	|	ε

AnyType	→	ANY
	|	ANY DEFINED BY `identifier`

ObjectIdentifierType	→	OBJECT IDENTIFIER

UsefulType	→	EXTERNAL
	|	"UTCTime"
	|	"GeneralizedTime"
	|	"ObjectDescriptor"

CharacterStringType	→	"NumericString"
	|	"PrintableString"
	|	"TeletexString"
	|	"T61String"

| “VideotexString”
| “VisibleString”
| “ISO646String”
| “IA5String”
| “GraphicString”
| “GeneralString”

Étape 1

NamedType → `lower`$_{id}$ Type
| Type

Élimination de la production ‘SelectionType’ car :
Type ⇒ BuiltInType ⇒ SelectionType

Type → `upper` [“.” `upper`$_{typ}$]
| BuiltInType
| SetType
| SequenceType
| SetOfType
| SequenceOfType
| SelectionType
| TaggedType
| NullType
| Type SubtypeSpec
| SET SIZE SubtypeSpec OF Type
| SEQUENCE SIZE SubtypeSpec OF Type

Expansion totale de *‘DefinedType’*.
Expansion partielle des productions *‘SetType’*, *‘SequenceType’*, *‘SetOfType’*, *‘SequenceOfType’*, *‘NullType’*, *‘SelectionType’* et *‘TaggedType’* de la règle *‘BuiltInType’*.
Expansion globale de ‘*Subtype*’ après l’expansion globale de ‘ParentType’ :
Subtype ⇒ ParentType SubtypeSpec ⇒ Type SubtypeSpec
Le but est de factoriser à gauche la règle *‘Type’* et de faire apparaître la double récurrence, cause d’ambiguïté.
Expansion globale de ‘SizeConstraint’ qui est déplacée dans la section (10.5.4).

BuiltInType → BOOLEAN
| INTEGER [“{” {NamedNumber “,” ...}$^{+}$ “}”]
| BIT STRING [“{” {NamedBit “,” ...}$^{+}$ “}”]
| OCTET STRING
| CHOICE “{” {NamedType “,” ...}$^{+}$ “}”
| ANY [DEFINED BY $\texttt{lower}_{id}$]
| OBJECT IDENTIFIER
| ENUMERATED “{” {NamedNumber “,” ...}$^{+}$ “}”
| REAL
| “NumericString”
| “PrintableString”
| “TeletexString”
| “T61String”
| “VideotexString”
| “VisibleString”
| “ISO646String”
| “IA5String”
| “GraphicString”
| “GeneralString”
| EXTERNAL
| “UTCTime”
| “GeneralizedTime”
| “ObjectDescriptor”

Expansion globale de *‘BooleanType’*, *‘IntegerType’*, *‘BitStringType’*, *‘OctetStringType’*, *‘TaggedType’*, *‘AnyType’*, *‘ObjectIdentifierType’*, *‘UsefulType’*, *‘CharacterStringType’*, *‘RealType’*.
Arden de ‘Enumeration’ et expansion globale, puis expansion globale de *‘EnumeratedType’*.
Arden de ‘AlternativeTypeList’ et expansion globale, puis expansion globale de *‘ChoiceType’*.
Cf. *‘Type’*, *‘NamedNumber’*, *‘NamedBit’*.

NamedNumber → $\texttt{lower}_{id}$ “(” AuxNamedNum “)”
AuxNamedNum → SignedNumber
| DefinedValue

Arden de ‘NamedNumberList’ et expansion globale.
Factorisation préfixe de *‘IntegerType’* puis expansion globale.
Factorisation bifixe de ‘NamedNumber’ (‘AuxNamedNum’).

NamedBit → $\texttt{lower}_{id}$ “(” ClassNumber “)”

Factorisation bifixe : nous reconnaissons la règle ‘ClassNumber’.

SequenceOfType → SEQUENCE [OF Type]
SetOfType → SET [OF Type]

Factorisations préfixes.

SequenceType → SEQUENCE “{” {ElementType “,” . . . }* “}”
SetType → SET “{” {ElementType “,” . . . }* “}”
ElementType → NamedType [ElementTypeSuf]
| COMPONENTS OF Type
ElementTypeSuf → OPTIONAL
| DEFAULT Value

Factorisation bifixe de *‘SequenceType’* et *‘SetType’*.
Arden de ‘ElementTypeList’ et expansion globale.
Factorisation préfixe de ‘ElementType’.

SelectionType → `lower`$_{id}$ “<” Type

NullType → NULL

TaggedType → “[” [Class] ClassNumber “]” [TagDefault] Type
Class → UNIVERSAL
| APPLICATION
| PRIVATE

Factorisation bifixe de *‘TaggedType’* : nous reconnaissons l’option de la règle *‘TagDefault’* (Cf. 10.5.1).
Expansion globale de ‘Tag’.
Option de ‘Class’.

ClassNumber → `number`
| [`upper`$_{mod}$ “.”] `lower`$_{val}$

Expansion totale de *'DefinedValue'*.

Étape 2

NamedType → `lower`$_{id}$ Type
| Type

Type → `upper` [“.” `upper`$_{typ}$]
| BuiltInType
| SET “{” {ElementType “,” . . . }* “}”
| SEQUENCE “{” {ElementType “,” . . . }* “}”
| SET [OF Type]
| SEQUENCE [OF Type]
| `lower`$_{id}$ “<” Type
| “[” [Class] ClassNumber “]” [TagDefault] Type
| NULL
| Type SubtypeSpec
| SET SIZE SubtypeSpec OF Type
| SEQUENCE SIZE SubtypeSpec OF Type

Expansion globale de *'SetType'*, *'SequenceType'*, *'SetOfType'*, *'SequenceOfType'*, *'SelectionType'*, *'TaggedType'* et *'NullType'*.

BuiltInType → BOOLEAN
| INTEGER [“{” {NamedNumber “,” . . . }$^+$ “}”]
| BIT STRING [“{” {NamedBit “,” . . . }$^+$ “}”]
| OCTET STRING
| CHOICE “{” {NamedType “,” . . . }$^+$ “}”
| ANY [DEFINED BY `lower`$_{id}$]
| OBJECT IDENTIFIER
| ENUMERATED “{” {NamedNumber “,” . . . }$^+$ “}”
| REAL
| “NumericString”
| “PrintableString”
| “TeletexString”
| “T61String”

| “VideotexString”
| “VisibleString”
| “ISO646String”
| “IA5String”
| “GraphicString”
| “GeneralString”
| EXTERNAL
| “UTCTime”
| “GeneralizedTime”
| “ObjectDescriptor”

NamedNumber → `lower`$_{id}$ “(” AuxNamedNum “)”
AuxNamedNum → [“-”] `number`
| [`upper`$_{mod}$ “.”] `lower`$_{val}$

Expansion totale de ‘SignedNumber’ et *‘DefinedValue’* dans ‘AuxNamedNum’.
Déplacement de ‘SignedNumber’ et *‘DefinedValue’* dans la section (10.5.3).

NamedBit → `lower`$_{id}$ “(” ClassNumber “)”

ElementType → NamedType [ElementTypeSuf]
| COMPONENTS OF Type
ElementTypeSuf → OPTIONAL
| DEFAULT Value

Class → UNIVERSAL
| APPLICATION
| PRIVATE
ClassNumber → `number`
| [`upper`$_{mod}$ “.”] `lower`$_{val}$

Étape 3

Nous supprimons ici l’ambiguïté de la règle *‘Type’*, en appliquant la transformation vue à la section 10.1.2 page 237. Nous rendrons ensuite LL(1) la règle *‘NamedType’*.

Type	→	`lower`$_{id}$ "<" Type
	|	"[" [Class] ClassNumber "]" [TagDefault] Type
	|	SetSeq [SIZE SubtypeSpec] OF Type
	|	Type SubtypeSpec
	|	NULL
	|	BuiltInType
	|	`upper` ["." `upper`$_{typ}$]
	|	SetSeq ["{" {ElementType "," ...}* "}"]
SetSeq	→	SET
	|	SEQUENCE

Factorisations suffixes ('SetSeq').

Type	→	`lower`$_{id}$ "<" Type
	|	"[" [Class] ClassNumber "]" [TagDefault] Type
	|	SetSeq [SIZE SubtypeSpec] OF Type
	|	NULL SubtypeSpec*
	|	BuiltInType SubtypeSpec*
	|	`upper` ["." `upper`$_{typ}$] SubtypeSpec*
	|	SetSeq ["{" {ElementType "," ...}* "}"] SubtypeSpec*
SetSeq	→	SET
	|	SEQUENCE

Application de la transformation vue à la section (10.1.2).

Type	→	`lower`$_{id}$ "<" Type
	|	"[" [Class] ClassNumber "]" [TagDefault] Type
	|	NULL SubtypeSpec*
	|	BuiltInType SubtypeSpec*
	|	`upper` ["." `upper`$_{typ}$] SubtypeSpec*
	|	SetSeq TypeSuf
SetSeq	→	SET
	|	SEQUENCE
TypeSuf	→	["{" {ElementType "," ...}* "}"] SubtypeSpec*
	|	[SIZE SubtypeSpec] OF Type

Factorisation préfixe (‘TypeSuf’).

Type	→	$\texttt{lower}_{id}$ “<” Type
	|	$\texttt{upper}$ [“.” $\texttt{upper}_{typ}$] SubtypeSpec*
	|	NULL SubtypeSpec*
	|	AuxType
AuxType	→	“[” [Class] ClassNumber “]” [TagDefault] Type
	|	BuiltInType SubtypeSpec*
	|	SetSeq [TypeSuf]
SetSeq	→	SET
	|	SEQUENCE
TypeSuf	→	SubtypeSpec+
	|	“{” {ElementType “,” . . . }* “}” SubtypeSpec*
	|	[SIZE SubtypeSpec] OF Type

Option de ‘TypeSuf’.
Réduction de *‘Type’* (‘AuxType’). On comprendra pourquoi à l’étape 4 de la section (10.5.3).

NamedType	→	$\texttt{lower}_{id}$ Type
	|	Type

NamedType	→	$\texttt{lower}_{id}$ Type
	|	$\texttt{lower}_{id}$ “<” Type
	|	$\texttt{upper}$ [“.” $\texttt{upper}_{typ}$] SubtypeSpec*
	|	NULL SubtypeSpec*
	|	AuxType

Expansion totale de *‘Type’*.

NamedType	→	$\texttt{lower}_{id}$ [“<”] Type
	|	$\texttt{upper}$ [“.” $\texttt{upper}_{typ}$] SubtypeSpec*
	|	NULL SubtypeSpec*
	|	AuxType

Factorisation bifixe.

Étape 4

Voici la grammaire finale de la section 10.5.2 page 297.

Type	→	`lower`$_{id}$ "<" Type
	|	`upper` ["." `upper`$_{typ}$] SubtypeSpec*
	|	NULL SubtypeSpec*
	|	AuxType
AuxType	→	"[" [Class] ClassNumber "]" [TagDefault] Type
	|	BuiltInType SubtypeSpec*
	|	SetSeq [TypeSuf]
SetSeq	→	SET
	|	SEQUENCE
TypeSuf	→	SubtypeSpec+
	|	"{" {ElementType "," ...}* "}" SubtypeSpec*
	|	[SIZE SubtypeSpec] OF Type

BuiltInType	→	BOOLEAN
	|	INTEGER ["{" {NamedNumber "," ...}+ "}"]
	|	BIT STRING ["{" {NamedBit "," ...}+ "}"]
	|	OCTET STRING
	|	CHOICE "{" {NamedType "," ...}+ "}"
	|	ANY [DEFINED BY `lower`$_{id}$]
	|	OBJECT IDENTIFIER
	|	ENUMERATED "{" {NamedNumber "," ...}+ "}"
	|	REAL
	|	"NumericString"
	|	"PrintableString"
	|	"TeletexString"
	|	"T61String"
	|	"VideotexString"
	|	"VisibleString"
	|	"ISO646String"
	|	"IA5String"
	|	"GraphicString"
	|	"GeneralString"
	|	EXTERNAL

		“UTCTime”
		“GeneralizedTime”
		“ObjectDescriptor”

NamedType	→	`lower`$_{id}$ [“<”] Type
			`upper` [“.” `upper`$_{typ}$] SubtypeSpec*
			NULL SubtypeSpec*
			AuxType

NamedNumber	→	`lower`$_{id}$ “(” AuxNamedNum “)”
AuxNamedNum	→	[“-”] `number`
			[`upper`$_{mod}$ “.”] `lower`$_{val}$

NamedBit	→	`lower`$_{id}$ “(” ClassNumber “)”

ElementType	→	NamedType [ElementTypeSuf]
			COMPONENTS OF Type
ElementTypeSuf	→	OPTIONAL
			DEFAULT Value

Class	→	UNIVERSAL
			APPLICATION
			PRIVATE
ClassNumber	→	`number`
			[`upper`$_{mod}$ “.”] `lower`$_{val}$

10.5.3 Transformations de la grammaire des valeurs

Étape 0

Voici la forme normalisée de la grammaire des valeurs ASN.1 Notons que nous récupérons ici les règles *‘ObjectIdentifierValue’*, en provenance de la section 10.5.1 page 292, *‘DefinedValue’* et *‘SignedNumber’*, en provenance de la section 10.5.2 page 297.

Value	→	BuiltInValue
	|	DefinedValue
DefinedValue	→	[`upper`$_{mod}$ "."] `lower`$_{val}$
SignedNumber	→	["-"] `number`
BuiltInValue	→	BooleanValue
	|	IntegerValue
	|	BitStringValue
	|	OctetStringValue
	|	NullValue
	|	SequenceValue
	|	SequenceOfValue
	|	SetValue
	|	SetOfValue
	|	ChoiceValue
	|	SelectionValue
	|	TaggedValue
	|	AnyValue
	|	ObjectIdentifierValue
	|	CharacterStringValue
	|	EnumeratedValue
	|	RealValue
NamedValue	→	`identifier` Value
	|	Value
BooleanValue	→	TRUE
	|	FALSE
IntegerValue	→	SignedNumber
	|	`identifier`
EnumeratedValue	→	`identifier`

RealValue	→	NumericRealValue
	|	SpecialRealValue
NumericRealValue	→	"{" Mantissa "," Base "," Exponent "}"
	|	"0"
Mantissa	→	SignedNumber
Base	→	"2"
	|	"10"
Exponent	→	SignedNumber
SpecialRealValue	→	PLUS-INFINITY
	|	MINUS-INFINITY

OctetStringValue	→	`bstring`
	|	`hstring`

BitStringValue	→	`bstring`
	|	`hstring`
	|	"{" IdentifierList "}"
	|	"{" "}"
IdentifierList	→	`identifier`
	|	IdentifierList "," `identifier`

NullValue	→	NULL

SequenceValue	→	"{" ElementValueList "}"
	|	"{" "}"
SetValue	→	"{" ElementValueList "}"
	|	"{" "}"
ElementValueList	→	NamedValue
	|	ElementValueList "," NamedValue

ChoiceValue	→	[`identifier` " :"] Value

Et non pas : *ChoiceValue* → NamedValue (**Erratum**).

SelectionValue → Value

Et non pas : *SelectionValue* → NamedValue (**Erratum**).

SequenceOfValue → “{” ValueList “}”
| “{” “}”
SetOfValue → “{” ValueList “}”
| “{” “}”
ValueList → Value
| ValueList “,” Value

TaggedValue → Value

AnyValue → Type “ :” Value

Et non pas : *AnyValue* → Type Value (**Erratum**).

ObjectIdentifierValue → “{” ObjIdComponent$^+$ “}”

CharacterStringValue → `cstring`

Étape 1

On prendra garde à ce que les terminaux `bstring` et `hstring` sont fusionnés en `basednumber`. Voir section 10.1.3 page 239. D'autre part, les productions en doublon ne sont pas immédiatement fusionnées pour des raisons de clarté. On notera aussi que les terminaux “0”, “2” et “10” deviennent le terminal `number`. (C'est ce que fait normalement un analyseur lexical.)

Value → BuiltInValue
| IntegerValue

| NullValue
| ChoiceValue
| SelectionValue
| TaggedValue
| AnyValue
| EnumeratedValue
| [`upper`$_{mod}$ “.”] `lower`$_{val}$

Expansion partielle de *‘IntegerValue’*, *‘NullValue’*, *‘ChoiceValue’*, *‘SelectionValue’*, *‘TaggedValue’*, *‘AnyValue’* et *‘EnumeratedValue’* de la règle *‘BuiltInValue’*.
Expansion globale de *‘DefinedValue’*.

BuiltInValue → TRUE
| FALSE
| `basednum`
| “{” {`lower`$_{id}$ “,” ...}* “}”
| `basednum`
| “{” {NamedValue “,” ...}* “}”
| “{” {Value “,” ...}* “}”
| “{” {NamedValue “,” ...}* “}”
| “{” {Value “,” ...}* “}”
| “{” ObjIdComponent^{+} “}”
| `string`
| “{” [“-”] `number` “,” `number` “,” [“-”] `number` “}”
| `number`
| PLUS-INFINITY
| MINUS-INFINITY

Expansion globale de *‘BooleanValue’*, *‘ObjectIdentifierValue’*, *‘CharacterStringValue’* et *‘OctetStringValue’*.
Arden de ‘IdentifierList’ et expansion globale, puis factorisation bifixe dans *‘BitStringValue’*.
Expansion globale de *‘BitStringValue’*.
Arden de ‘ElementValueList’ et expansion globale, puis factorisation bifixe dans *‘SequenceValue’* et *‘SetValue’*.
Expansion globale de *‘SequenceValue’* et *‘SetValue’*.
Arden de ‘ValueList’ et expansion globale, puis factorisation bifixe dans *‘SequenceOfValue’* et *‘SetOfValue’*.
Expansion globale de *‘SequenceOfValue’* et *‘SetOfValue’*.
Sous-section *‘RealValue’* : expansion totale de *‘SignedNumber’* dans ‘Mantissa’ et ‘Exponent’.
Expansion globale de ‘Mantissa’, ‘Base’ (Base → `number`) et ‘Exponent’.
Expansion globale de ‘NumericRealValue’, ‘SpecialRealValue’ et *‘RealValue’*.

NamedValue	→	`lower`$_{id}$ Value
	|	Value

IntegerValue	→	[“-”] `number`
	|	`lower`$_{id}$

Expansion globale de *‘SignedNumber’*.

EnumeratedValue	→	`lower`$_{id}$

NullValue	→	NULL

ChoiceValue	→	[`lower`$_{id}$ “ :”] Value

SelectionValue	→	Value

TaggedValue	→	Value

AnyValue	→	Type “ :” Value

Étape 2

Value	→	BuiltInValue
	|	[“-”] `number`
	|	`lower`$_{id}$
	|	NULL
	|	[`lower`$_{id}$ “ :”] Value
	|	Value
	|	Value

	Type " :" Value
|	`lower`$_{id}$
|	[`upper`$_{mod}$ "."] `lower`$_{val}$
|	`number`

Expansion globale de *'IntegerValue'*, *'NullValue'*, *'ChoiceValue'*, *'SelectionValue'*, *'TaggedValue'*, *'AnyValue'*, et *'EnumeratedValue'*.
Expansion partielle de *BuiltInValue* → `number`

BuiltInValue	→	TRUE
	|	FALSE
	|	PLUS-INFINITY
	|	MINUS-INFINITY
	|	`basednum`
	|	`string`
	|	"{" BetBraces "}"
BetBraces	→	{NamedValue "," ...}*
	|	ObjIdComponent^{+}
	|	["-"] `number` "," `number` "," ["-"] `number`

Lemme d'Arden (Élimination des redondances Value → Value | ...).
Élimination de la production "{" {`lower`$_{id}$ "," ...}* "}"
car "{" {Value "," ...}* "}" ⇒ "{" {`lower`$_{id}$ "," ...}* "}"
Élimination de la production "{"{Value "," ...}* "}"
car "{" {NamedValue "," ...}* "}" ⇒ "{"{Value "," ...}* "}"
Factorisation bifixe ('BetBraces').

NamedValue	→	`lower`$_{id}$ Value
	|	Value

Étape 3

Value	→	BuiltInValue
	|	["-"] `number`
	|	NULL
	|	`lower` [" :" Value]

	| Type " :" Value
	| `upper`$_{mod}$ "." `lower`$_{val}$

Élimination des redondances.
Arden de *'Value'*.
Factorisation préfixe de `lower`.

BuiltInValue	→	TRUE
	|	FALSE
	|	PLUS-INFINITY
	|	MINUS-INFINITY
	|	`basednum`
	|	`string`
	|	"{" [BetBraces] "}"
BetBraces	→	{NamedValue "," ...}$^{+}$
	|	ObjIdComponent^{+}

Option de 'BetBraces'.
Élimination de la troisième production de 'BetBraces' car :
{NamedValue "," ...}$^{+} \overset{+}{\Longrightarrow}$ ["-"] `number` "," `number` "," ["-"] `number`

NamedValue	→	`lower`$_{id}$ Value
	|	Value

Étape 4

Nous rendons LL(1) la règle *'Value'*, puis *'NamedValue'*.

Value	→	BuiltInValue
	|	["-"] `number`
	|	NULL
	|	`lower` [" :" Value]
	|	`upper`$_{mod}$ "." `lower`$_{val}$
	|	`lower`$_{id}$ "<" Type " :" Value
	|	`upper` [" ." `upper`$_{typ}$] SubtypeSpec* " :" Value
	|	NULL SubtypeSpec* " :" Value
	|	AuxType " :" Value

Expansion totale de *'Type'*.

Value	→	AuxVal0
	|	`upper` AuxVal1
	|	`lower` [AuxVal2]
	|	["-"] `number`
AuxVal0	→	BuiltInValue
	|	AuxType " :" Value
	|	NULL [SubtypeSpec* " :" Value]
AuxVal1	→	["." $\texttt{upper}_{typ}$] SubtypeSpec* " :" Value
	|	"." $\texttt{lower}_{val}$
AuxVal2	→	["<" Type] " :" Value

Factorisations préfixes.
Réduction ('AuxVal0').
La raison est donnée en (10.5.3).

Value	→	AuxVal0
	|	`upper` AuxVal1
	|	`lower` [AuxVal2]
	|	["-"] `number`
AuxVal0	→	BuiltInValue
	|	AuxType " :" Value
	|	NULL [SpecVal]
AuxVal1	→	SpecVal
	|	"." AuxVal11
AuxVal2	→	["<" Type] " :" Value
AuxVal11	→	$\texttt{upper}_{typ}$ SpecVal
	|	$\texttt{lower}_{val}$
SpecVal	→	SubtypeSpec* " :" Value

Expansion inverse ('SpecVal').
Factorisation préfixe de 'AuxVal1' ('AuxVal11').

NamedValue	→	$\texttt{lower}_{id}$ Value

	AuxVal0
|	`upper` AuxVal1
|	`lower` [AuxVal2]
|	[“-”] `number`

Expansion *‘Value’*.

NamedValue	→	`lower` [NamedValSuf]
	|	`upper` AuxVal1
	|	[“-”] `number`
	|	AuxVal0
NamedValSuf	→	Value
	|	AuxVal2

Factorisation préfixe.

Étape 5

Nous rendons maintenant LL(1) la règle ‘BetBraces’.

BetBraces	→	NamedValue [“,” {NamedValue “,” ...}$^+$]
	|	ObjIdComponent ObjIdComponent*

Opérations rationnelles.

BetBraces	→	`lower` [NamedValSuf] [AuxNamed]
	|	`upper` AuxVal1 [AuxNamed]
	|	[“-”] `number` [AuxNamed]
	|	AuxVal0 [AuxNamed]
	|	`number` ObjIdComponent*
	|	`upper`$_{mod}$ “.” `lower`$_{val}$ ObjIdComponent*
	|	`lower` [“(” ClassNumber “)”] ObjIdComponent*
AuxNamed	→	“,” {NamedValue “,” ...}$^+$

Expansion inverse (‘AuxNamed’).
Expansion totale de *‘NamedValue’*.
Expansion totale de *‘ObjIdComponent’*.
Cf. (10.5.1).

BetBraces → AuxVal0 [AuxNamed]
| “-” `number` [AuxNamed]
| `lower` [AuxBet1]
| `upper` AuxBet2
| `number` [AuxBet3]

AuxBet1 → “(” ClassNumber “)” ObjIdComponent*
| ObjIdComponent+
| NamedValSuf [AuxNamed]
| AuxNamed

AuxBet2 → AuxVal1 [AuxNamed]
| “.” $\texttt{lower}_{val}$ ObjIdComponent*

AuxBet3 → ObjIdComponent+
| AuxNamed

Factorisations préfixes et développement de ‘AuxBet1’.

AuxBet1 → “(” ClassNumber “)” ObjIdComponent*
| AuxNamed
| ObjIdComponent ObjIdComponent*
| ObjIdComponent ObjIdComponent*
| ObjIdComponent ObjIdComponent*
| Value [AuxNamed]
| AuxVal2 [AuxNamed]

AuxBet2 → SpecVal [AuxNamed]
| “.” $\texttt{upper}_{typ}$ SpecVal [AuxNamed]
| “.” $\texttt{lower}_{val}$ [AuxNamed]
| “.” $\texttt{lower}_{val}$ ObjIdComponent*

Opération rationnelle dans ‘AuxBet1’.
Expansion totale de ‘NamedValSuf’.
Expansion totale de ‘AuxVal1’ dans ‘AuxBet2’, puis de ‘AuxVal11’.

AuxBet1 → "(" ClassNumber ")" ObjIdComponent*
| AuxNamed
| AuxVal2 [AuxNamed]
| `number` ObjIdComponent*
| `upper`$_{mod}$ "." `lower`$_{val}$ ObjIdComponent*
| `lower` ["(" ClassNumber ")"] ObjIdComponent*
| AuxVal0 [AuxNamed]
| `upper` AuxVal1 [AuxNamed]
| `lower` [AuxVal2] [AuxNamed]
| ["-"] `number` [AuxNamed]

AuxBet2 → SpecVal [AuxNamed]
| "." AuxBet21

AuxBet21 → `upper`$_{typ}$ SpecVal [AuxNamed]
| `lower`$_{val}$ [AuxBet3]

Expansion totale de 'ObjectIdComponent'.
Expansion totale de '*Value*'.
Factorisation préfixe de 'AuxBet2'.
Nous reconnaissons 'AuxBet3'.

AuxBet1 → "(" ClassNumber ")" ObjIdComponent*
| AuxNamed
| AuxVal2 [AuxNamed]
| "-" `number` [AuxNamed]
| AuxVal0 [AuxNamed]
| `lower` [AuxBet11]
| `number` [AuxBet3]
| `upper` AuxBet2

AuxBet11 → "(" ClassNumber ")" ObjIdComponent*
| ObjIdComponent^{+}
| AuxVal2 [AuxNamed]
| AuxNamed

Factorisations préfixes dans 'AuxBet1' ('AuxBet11') où nous reconnaissons 'AuxBet3' et 'AuxBet2'.
Nous développons 'AuxBet11'.
NOTA : Nous n'avons pas fait figurer ici les règles 'AuxBet2' et 'AuxBet21'.

Étape 6

Nous rappelons le bilan des transformations.

Value	→	AuxVal0
	|	`upper` AuxVal1
	|	`lower` [AuxVal2]
	|	["-"] `number`
AuxVal0	→	BuiltInValue
	|	AuxType " :" Value
	|	NULL [SpecVal]
AuxVal1	→	SpecVal
	|	"." AuxVal11
AuxVal2	→	["<" Type] " :" Value
AuxVal11	→	$\texttt{upper}_{typ}$ SpecVal
	|	$\texttt{lower}_{val}$
SpecVal	→	SubtypeSpec* " :" Value

BuiltInValue	→	TRUE
	|	FALSE
	|	PLUS-INFINITY
	|	MINUS-INFINITY
	|	`basednum`
	|	`string`
	|	"{" [BetBraces] "}"

BetBraces	→	AuxVal0 [AuxNamed]
	|	"-" `number` [AuxNamed]
	|	`lower` [AuxBet1]
	|	`upper` AuxBet2
	|	`number` [AuxBet3]
AuxBet1	→	"(" ClassNumber ")" ObjIdComponent*
	|	AuxNamed
	|	AuxVal2 [AuxNamed]
	|	"-" `number` [AuxNamed]
	|	AuxVal0 [AuxNamed]
	|	`lower` [AuxBet11]
	|	`number` [AuxBet3]

	|	`upper` AuxBet2
AuxBet2	→	SpecVal [AuxNamed]
	|	"." AuxBet21
AuxBet3	→	ObjIdComponent$^+$
	|	AuxNamed
AuxBet11	→	"(" ClassNumber ")" ObjIdComponent*
	|	ObjIdComponent$^+$
	|	AuxVal2 [AuxNamed]
	|	AuxNamed
AuxBet21	→	`upper`$_{typ}$ SpecVal [AuxNamed]
	|	`lower`$_{val}$ [AuxBet3]
AuxNamed	→	"," {NamedValue "," ...}$^+$
NamedValue	→	`lower` [NamedValSuf]
	|	`upper` AuxVal1
	|	["-"] `number`
	|	AuxVal0
NamedValSuf	→	Value
	|	AuxVal2

10.5.4 Transformations de la grammaire des sous-types

Étape 0

Nous présentons ici la forme normalisée de la grammaire des sous-types ASN.1, avec un découpage en sous-sections. Notons que nous récupérons ici la règle 'SizeConstraint' en provenance de la section 10.5.2 page 297.

SubtypeSpec	→	"(" SubtypeValueSet SubtypeValueSetList ")"
SubtypeValueSetList	→	"|" SubtypeValueSet SubtypeValueSetList
	|	ε
SubtypeValueSet	→	SingleValue
	|	ContainedSubtype
	|	ValueRange
	|	PermittedAlphabet
	|	SizeConstraint
	|	InnerTypeConstraints

SingleValue	→	Value

ContainedSubtype	→	INCLUDES Type
ValueRange	→	LowerEndPoint ".." UpperEndPoint
LowerEndPoint	→	LowerEndValue
	|	LowerEndValue "<"
UpperEndPoint	→	UpperEndValue
	|	"<" UpperEndValue
LowerEndValue	→	Value
	|	MIN
UpperEndValue	→	Value
	|	MAX
SizeConstraint	→	SIZE SubtypeSpec
PermittedAlphabet	→	FROM SubtypeSpec
InnerTypeConstraints	→	WITH COMPONENT SingleTypeConstraint
	|	WITH COMPONENTS MultipleTypeConstraints
SingleTypeConstraint	→	SubtypeSpec
MultipleTypeConstraints	→	FullSpecification
	|	PartialSpecification
FullSpecification	→	"{" TypeConstraints "}"
PartialSpecification	→	"{" "..." "," TypeConstraints "}"
TypeConstraints	→	NamedConstraint
	|	NamedConstraint "," TypeConstraints
NamedConstraint	→	`identifier` Constraint
	|	Constraint
Constraint	→	ValueConstraint PresenceConstraint
ValueConstraint	→	SubtypeSpec
	|	ε
PresenceConstraint	→	PRESENT
	|	ABSENT
	|	OPTIONAL
	|	ε

Étape 1

SubtypeSpec	→	"(" {SubtypeValueSet "|" ...}$^+$ ")"
SubtypeValueSet	→	SingleValue
	|	ContainedSubtype
	|	ValueRange
	|	PermittedAlphabet
	|	SizeConstraint
	|	InnerTypeConstraints

Arden de 'SubtypeValueSetList', puis expansion globale.

SingleValue	→	Value

ContainedSubtype	→	INCLUDES Type

ValueRange	→	LowerEndValue ["<"] ".." ["<"] UpperEndValue
LowerEndValue	→	Value
	|	MIN
UpperEndValue	→	Value
	|	MAX

Factorisation préfixe de 'LowerEndPoint'.
Factorisation suffixe de 'UpperEndPoint'.
Expansion globale de 'LowerEndPoint' et 'UpperEndPoint'.

SizeConstraint	→	SIZE SubtypeSpec

PermittedAlphabet	→	FROM SubtypeSpec

InnerTypeConstraints	→	WITH InnerTypeSuf
InnerTypeSuf	→	COMPONENT SubtypeSpec
	|	COMPONENTS MultipleTypeConstraints

MultipleTypeConstraints	→	“{” [“. . . ” “,”] TypeConstraints “}”
TypeConstraints	→	{NamedConstraint “,” . . . }$^+$
NamedConstraint	→	[`lower`$_{id}$] Constraint
Constraint	→	[SubtypeSpec] [PresenceConstraint]
PresenceConstraint	→	PRESENT
	|	ABSENT
	|	OPTIONAL

Expansion globale de ‘SingleTypeConstraint’.
Factorisation préfixe de *‘InnerTypeConstraints’* (‘InnerTypeSuf’).
Expansion globale de ‘FullSpecification’ et ‘PartialSpecification’.
Factorisation bifixe de ‘MultipleTypeConstraints’.
Arden de ‘TypeConstraints’.
Factorisation suffixe de ‘NamedConstraint’.
Option de ‘ValueConstraint’, puis expansion globale.
Option de ‘PresenceConstraint’.

Étape 2

SubtypeSpec	→	“(” {SubtypeValueSet “|” . . . }$^+$ “)”
SubtypeValueSet	→	Value
	|	INCLUDES Type
	|	LowerEndValue [“<”] “..” [“<”] UpperEndValue
	|	FROM SubtypeSpec
	|	SIZE SubtypeSpec
	|	WITH InnerTypeSuf
LowerEndValue	→	Value
	|	MIN
UpperEndValue	→	Value
	|	MAX
InnerTypeSuf	→	COMPONENT SubtypeSpec
	|	COMPONENTS MultipleTypeConstraints
MultipleTypeConstraints	→	“{” [“. . . ” “,”] {[NamedConstraint] “,” . . . } “}”
NamedConstraint	→	`lower`$_{id}$ [SubtypeSpec] [PresenceConstraint]
	|	SubtypeSpec [PresenceConstraint]
	|	PresenceConstraint
PresenceConstraint	→	PRESENT
	|	ABSENT
	|	OPTIONAL

Expansion globale de ‘SingleValue’.
Expansion globale de ‘ContainedSubtype’.
Expansion globale de ‘ValueRange’.
Expansion globale de ‘PermittedAlphabet’.
Expansion globale de ‘SizeConstraint’.
Expansion globale de ‘InnerTypeConstraints’.
Expansion globale de ‘TypeConstraints’.
Expansion globale de ‘Constraint’ puis option de ‘NamedConstraint’.

Étape 3

SubtypeSpec	→	“(” {SubtypeValueSet “|” . . . }$^+$ “)”
SubtypeValueSet	→	Value [SubValSetSuf]
		| INCLUDES Type
		| MIN SubValSetSuf
		| FROM SubtypeSpec
		| SIZE SubtypeSpec
		| WITH InnerTypeSuf
SubValSetSuf	→	[“<”] “..” [“<”] UpperEndValue
UpperEndValue	→	Value
		| MAX
InnerTypeSuf	→	COMPONENT SubtypeSpec
		| COMPONENTS MultipleTypeConstraints
MultipleTypeConstraints	→	“{” [“. . . ” “,”] {[NamedConstraint] “,” . . . } “}”
NamedConstraint	→	`lower`$_{id}$ [SubtypeSpec] [PresenceConstraint]
		| SubtypeSpec [PresenceConstraint]
		| PresenceConstraint
PresenceConstraint	→	PRESENT
		| ABSENT
		| OPTIONAL

Expansion globale de ‘LowerEndValue’.
Factorisation préfixe de ‘SubtypeValueSet’ (‘SubValSetSuf’).

Étape 4

Nous voyons apparaître un problème plus subtil qui empêche la grammaire d'être LL(1). En effet, nous avions obtenu précédemment (section 10.5.3 page 309) :

Value	→	AuxVal0
	|	`upper` AuxVal1
	|	`lower` [AuxVal2]
	|	["-"] `number`
AuxVal2	→	["<" Type] " :" Value

La troisième production de *'Value'* implique que le terminal "<" ne doit pas être un symbole possible après une occurrence de *'Value'*, sinon nous ne vérifions pas la troisième condition définissant une grammaire LL(1) — section 10.3 page 246. Or nous venons de former :

SubtypeValueSet	→	Value [SubValSetSuf]
	|	INCLUDES Type
	|	MIN SubValSetSuf
	|	FROM SubtypeSpec
	|	SIZE SubtypeSpec
	|	WITH InnerTypeSuf
SubValSetSuf	→	["<"] ".." ["<"] UpperEndValue

Ce qui implique que "<" *est* un symbole possible après *'Value'*... Nous allons montrer que nous pouvons éliminer cet obstacle. L'idée consiste à faire apparaître le sous-mot 'Value [SubValSetSuf]' partout où c'est possible à l'aide d'expansions, et à en former une règle. Nous tenterons ensuite de transformer cette règle pour en obtenir une définition ne produisant jamais un mot contenant 'Value [SubValSetSuf]' : à chaque occurrence (dans les règles dépendantes ou dans la règle elle-même) nous placerons un appel à cette règle. Il faut noter qu'il n'était pas certain *a priori* qu'une telle solution existait.

SubtypeValueSet	→	SVSAux
	|	INCLUDES Type
	|	MIN SubValSetSuf
	|	FROM SubtypeSpec
	|	SIZE SubtypeSpec
	|	WITH InnerTypeSuf
SubValSetSuf	→	["<"] ".." ["<"] UpperEndValue

SVSAux → Value [SubValSetSuf]

Factorisation inverse (‘SVSAux’).

SVSAux → AuxVal0 [SubValSetSuf]
| `upper` AuxVal1 [SubValSetSuf]
| `lower` [AuxVal2] [SubValSetSuf]
| [“-”] `number` [SubValSetSuf]

Expansion totale de *‘Value’*.

SVSAux → BuiltInValue [SubValSetSuf]
| AuxType “ :” Value [SubValSetSuf]
| NULL [SpecVal] [SubValSetSuf]
| `upper` SVSAux1
| `lower` [SVSAux2]
| [“-”] `number` [SubValSetSuf]

SVSAux1 → AuxVal1 [SubValSetSuf]

SVSAux2 → AuxVal2 [SubValSetSuf]
| SubValSetSuf

Expansion totale de ‘AuxVal0’.
Expansions inverses (‘SVSAux1’ et ‘SVSAux2’).

SVSAux → BuiltInValue [SubValSetSuf]
| AuxType “ :” SVSAux
| NULL [SVSAux3]
| `upper` SVSAux1
| `lower` [SVSAux2]
| [“-”] `number` [SubValSetSuf]

SVSAux1 → SpecVal [SubValSetSuf]
| “.” AuxVal11 [SubValSetSuf]

SVSAux2 → [“<” Type] “ :” Value [SubValSetSuf]
| [“<”] “..” [“<”] UpperEndValue

SVSAux3 → SpecVal [SubValSetSuf]
| SubValSetSuf

Expansion inverse (‘SVSAux3’).
Expansion totale de ‘AuxVal1’.
Expansion totale de ‘AuxVal2’ et ‘SubValSetSuf’.
Expansions inverses (‘SVSAux1’ et ‘SVSAux2’).

SVSAux	→	BuiltInValue [SubValSetSuf]
	|	AuxType “ :” SVSAux
	|	NULL [SVSAux3]
	|	`upper` SVSAux1
	|	`lower` [SVSAux2]
	|	[“-”] `number` [SubValSetSuf]
SVSAux1	→	SubtypeSpec* “ :” Value [SubValSetSuf]
	|	“.” SVSAux11
SVSAux2	→	“ :” SVSAux
	|	“..” [“<”] UpperEndValue
	|	“<” SVSAux21
SVSAux3	→	SubtypeSpec* “ :” Value [SubValSetSuf]
	|	SubValSetSuf
SVSAux11	→	$\texttt{upper}_{typ}$ SpecVal [SubValSetSuf]
	|	$\texttt{lower}_{val}$ [SubValSetSuf]
SVSAux21	→	Type “ :” SVSAux
	|	“..” [“<”] UpperEndValue

Nous reconnaissons ‘SVSAux’ dans ‘SVSAux2’, puis factorisation préfixe (‘SVSAux21’).
Expansion totale de ‘SpecVal’ dans ‘SVSAux1’ et ‘SVSAux3’.
Expansion totale inverse de ‘SVSAux11’ (‘SVSAux11’) :
SVSAux11 → AuxVal11 [SubValSetSuf]
puis expansion totale de ‘AuxVal11’ dans ‘SVSAux11’.

SVSAux	→	BuiltInValue [SubValSetSuf]
	|	AuxType “ :” SVSAux
	|	NULL [SVSAux3]
	|	`upper` SVSAux1
	|	`lower` [SVSAux2]
	|	[“-”] `number` [SubValSetSuf]
SVSAux1	→	SubtypeSpec* “ :” SVSAux
	|	“.” SVSAux11
SVSAux2	→	“ :” SVSAux

	|	".." ["<"] UpperEndValue
	|	"<" SVSAux21
SVSAux3	→	SubtypeSpec* " :" SVSAux
	|	SubValSetSuf
SVSAux11	→	`upper`$_{typ}$ SubtypeSpec* " :" SVSAux
	|	`lower`$_{val}$ [SubValSetSuf]
SVSAux21	→	Type " :" SVSAux
	|	".." ["<"] UpperEndValue

Expansion totale de 'SpecVal' dans 'SVSAux11'.
Nous reconnaissons 'SVSAux' dans 'SVSAux1', 'SVSAux3' et 'SVSAux11'.

Étape 5

Nous présentons ici le bilan des transformations.

SubtypeSpec	→	"(" {SubtypeValueSet "|" ...}$^+$ ")"
SubtypeValueSet	→	INCLUDES Type
	|	MIN SubValSetSuf
	|	FROM SubtypeSpec
	|	SIZE SubtypeSpec
	|	WITH InnerTypeSuf
	|	SVSAux

InnerTypeSuf	→	COMPONENT SubtypeSpec
	|	COMPONENTS MultipleTypeConstraints
MultipleTypeConstraints	→	"{" ["..." ","] {[NamedConstraint] "," ...} "}"
NamedConstraint	→	`lower`$_{id}$ [SubtypeSpec] [PresenceConstraint]
	|	SubtypeSpec [PresenceConstraint]
	|	PresenceConstraint
PresenceConstraint	→	PRESENT
	|	ABSENT
	|	OPTIONAL

SubValSetSuf	→	["<"] ".." ["<"] UpperEndValue
UpperEndValue	→	Value

	|	MAX

SVSAux	→	BuiltInValue [SubValSetSuf]
	|	AuxType " :" SVSAux
	|	NULL [SVSAux3]
	|	`upper` SVSAux1
	|	`lower` [SVSAux2]
	|	["-"] `number` [SubValSetSuf]
SVSAux1	→	SubtypeSpec* " :" SVSAux
	|	"." SVSAux11
SVSAux2	→	" :" SVSAux
	|	".." ["<"] UpperEndValue
	|	"<" SVSAux21
SVSAux3	→	SubtypeSpec* " :" SVSAux
	|	SubValSetSuf
SVSAux11	→	$\texttt{upper}_{typ}$ SubtypeSpec* " :" SVSAux
	|	$\texttt{lower}_{val}$ [SubValSetSuf]
SVSAux21	→	Type " :" SVSAux
	|	".." ["<"] UpperEndValue

Bibliographie

A. Aho et J. Ullman : *Theory of parsing, Translation and Compiling*, volume 1 (Parsing) de *Automatic Computation*, chapitre 2.6.3, page 199. Prentice Hall, 1972.

A. Aho, R. Sethi et J. Ullman : *Compilers : principles, techniques and tools*. Addison-Wesley, 1986. ISBN 0-201-10088-6.

S. Bapat : *Object-Oriented Networks. Models for Architecture, Operations and Management*, chapitre 10 – Introduction to Network Modelling Syntax. PTR Prentice-Hall, 1994.

M. Bevar et U. Schäffer : Coding Rules for High Speed Networks. *In* G. Neufeld et B. Plattner, éditeurs : *Upper Layer Protocols, Architectures and Applications*, pages 119–132, 1992.

M. Bilgic et B. Sarikaya : Performance comparison of ASN.1 encode/decoders using FTAM. *In Computer Communication*, volume 16 (4), pages 229–240, avril 1993.

T. Blackwell : Fast Decoding of Tagged Message Formats. *In Conference on Computer Communications (IEEE Infocom)*, pages 224–231, San Francisco, California, USA, mars 1996. IEEE.

Gregor V. Bochmann : Usage of Protocol Development Tools : The Results of a Survey. *In* H. Rudin et C.H. West, éditeurs : *Seventh International Symposium on Protocol Specification, Testing and Verification (PSTV)*, pages 139–161, Zürich, Suisse, 1987.

Gregor V. Bochmann et Michel Deslauriers : Combining ASN.1 support with the LOTOS language. *In Nenth International Symposium on Protocol Specification, Testing and Verification (PSTV)*, pages 175–187, Enschede, Pays Bas, juin 1989.

Wayne Brookes et Jadwiga Indulska : A Formal Specification for a Basic ODP-based Type Manager. {brookes,jaga}@cs.uq.oz.au.

Anne-Marie Bustos : *Un environnement OSI autour du compilateur ASN.1 MAVROS*. Thèse de doctorat, Université de Nice Sophia-Antipolis, avril 1992. Préparée à l'INRIA Sophia-Antipolis.

Bruno Chatras : Spécification des règles d'encodage spécifiques pour la signalisation (SER). NT/DAC/SSR/12, Centre National d'Étude des Télécommunications (CNET, France Télécom), juillet 1997. 7 pages.

Anne-Marie Dery, Christian Huitema et William Maille : Environnement de programmation pour ASN.1. RR 1847, INRIA Sophia-Antipolis, février 1993. 29 pages.

E. Dillon : Protocoles de communication et représentation des données. ASN.1 et les langages de spécification formelle. Rapport technique, Université Nancy I, CRIN, septembre 1993. 46 pages.

Olivier Dubuisson et Joaquín Keller : Description formelle en Z de la sémantique statique de GDMO. DT/LAA/EIA/AIA/94-07, Centre National d'Étude des Télécommunications (CNET, France Télécom), mars 1994.

F. Duwez et Olivier Dubuisson : Implémentation d'un analyseur syntaxique descendant pour ASN.1 :1994. RP/LAA/EIA/123, Centre National d'Étude des Télécommunications (CNET, France Télécom), juillet 1996.

ETSI : Signalling Protocols and Switching (SPS) ; Evaluation of Abstract Syntax Notation One (ASN.1) tools for use as syntax and semantics checker. ETR 060, Référence DTR/SPS-2001, European Telecommunications Standards Institute, 06921 Sophia Antipolis cedex – France, novembre 1992.

European Telecommunications Standards Institute (ETSI) : Analysis of the use of ASN.1 :94 with TTCN and SDL in ETSI delivrables. DTR/MTS-47, ETSI, septembre 1997a. MTS (Methods for Testing and Specification).

European Telecommunications Standards Institute (ETSI) : ASN.1 :94 Presentation. TC MTS, PT 96, TD 44, ETSI, juin 1997b. 20 pages.

J. Fischer et R. Schröder : Combined Specification Using SDL and ASN.1. *In SDL'93*, pages 293–304, 1993.

Philippe Fouquart, F. Duwez et Olivier Dubuisson : Une analyse syntaxique d'ASN.1 :1994. RP/LAA/EIA/83, Centre National d'Étude des Télécommunications (CNET, France Télécom), février 1996.

M. Gardie : La couche présentation. La syntaxe ASN.1. Rapport technique, Institut National des Télécommunications (INT), février 1993.

C.W. Gardiner : ASN_EZE : An Analgesic for Writers of ASN.1 Applications. *In Software – Practice and Experience*, volume 26 (10), pages 1087–1096, octobre 1996.

P. Gaudette : A Tutorial on ASN.1. Rapport technique, NCSL/SNA-89/12, mai 1989. 36 pages.

P. Gaudette, S. Trus et S. Collins : An Object-Oriented Model for ASN.1. *In FORTE'88*, pages 121–134, 1988.

P. Gaudette, S. Trus et S. Collins : The Free Value Tool for ASN.1. NCSI/SNA-89/1, U.S. Dept. of Commerce, National Institute of Standards and Technology, janvier 1989. 40 pages.

W. Gora : ASN.1 Stand und Trends. *In Datacom*, volume 2, pages 114–122, 1990. En allemand.

Per Gunniberg, Erik Nordman, Stephen Pink, Peter Sjodin et Jan Erik Stromquist : Application protocols and performance benchmarks. *In IEEE Communications Magazine*, volume 27, pages 30–36, juin 1989.

K.W. Hart, D.B. Searls et G.C. Overton : SORTEZ : A Relational Translator for NCBI's ASN.1 Database. *In CABIOS*, volume 10 (4), pages 369–378, 1994.

K. Harumoto, M. Tsukamoto et S.Nishio : "Design and Implementation of the Compiler for an ASN.1 Database on OODBMS". Rapport technique, Dept. of Information Systems Engineering, Osaka University. 8 pages.

James D. Harvey et Alfred C. Weaver : Experience with the Abstract Syntax Notation One and the Basic Encoding Rules. *In IEEE Conference on Local Computer Networks*, pages 621–628, Minneapolis, Minnesota, USA, 1991.

T. Hasegawa, H. Horiuchi, T. Kato, K. Suzuki et Y. Urano : Automatic ADA Program Generation from Protocol Specification based on ESTELLE and ASN.1. *In International Conference on Computers and Communications*, pages 181–185, Tel Aviv, Israel, 1988.

H. Herzog, Karner, Heinz Schaefer et Schermann : An Adaptable Architecture for an ASN.1/TTCN/GDMO Tool Set. *In ISS'95*, 1995.

T.R. Hines : Open Systems Interconnection Abstract Notation : ASN.1, 1989. Univ. of Missouri, p. 303–311.

Philipp Hoschka : Towards tailoring protocols to application specific requirements. *In IEEE Infocom'93, Twelve annual joint conference of the IEEE Computer and Communication Societies*, volume 2, pages 647–653. IEEE, mai 1993a.

Philipp Hoschka : Towards tailoring protocols to applications specific requirements. *In Conference on Computer Communications (IEEE Infocom)*, pages 647–653, San Francisco, California, USA, avril 1993b. IEEE.

Philipp Hoschka : Use of ASN.1 for Remote Procedure Call Interfaces. *In Interop Europe*, Paris, octobre 1993c. 12 pages.

Philipp Hoschka : Automatic Performance Optimisation by Heuristic Analysis of a Formal Secification. *In* R. Gotzhein et J. Bredereke, éditeurs : *Formal Description Techniques IX*, pages 77–92. Kaiserslläutern, 1996.

Ch. Huitema et G. Chave : Measuring the performance of an ASN.1 compiler. *In* G. Neufeld et B. Plattner, éditeurs : *Upper Layer Protocols, Architectures and Applications*, pages 105–118, 1992.

Ch. Huitema et A. Doghri : Defining Faster Transfer Syntaxes for the OSI Presentation Protocol. *In ACM SIGCOMM Computer Communication Review*, volume 19 (5), pages 44–55, 1989.

Information processing systems – Text communications – Remote Operations – Part 1 : Model, notation and service definition. ISO/IEC, first édition, novembre 1989. Reference ISO/IEC 9072-1.

Information technology – Open Systems Interconnection – Specification of Abstract Syntax Notation One (ASN.1). ITU-T Recommendation

X.208, seconde édition, décembre 1990. Référence ISO/IEC 8824 :1990 (E).

Information technology – Abstract Syntax Notation One (ASN.1) : Specification of basic notation. ITU-T Recommendation X.680, juillet 1994a. Référence ISO/IEC 8824-1 :1994 (E).

Information technology – Abstract Syntax Notation One (ASN.1) : Information object specification. ITU-T Recommendation X.681, juillet 1994b. Référence ISO/IEC 8824-2 :1995 (E).

Information technology – Abstract Syntax Notation One (ASN.1) : Constraint specification. ITU-T Recommendation X.682, juillet 1994c. Référence ISO/IEC 8824-3 :1995 (E).

Information technology – Abstract Syntax Notation One (ASN.1) : Parametrization of ASN.1 specifications. ITU-T Recommendation X.683, juillet 1994d. Référence ISO/IEC 8824-4 :1995 (E).

Information technology – ASN.1 Encoding Rules : Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER). ITU-T Recommendation X.690, juillet 1994e. Référence ISO/IEC 8825-1 :1995 (E).

Information technology – ASN.1 Encoding Rules : Specification of Packed Encoding Rules (PER). ITU-T Recommendation X.691, juillet 1994f. Référence ISO/IEC 8825-2 :1995 (E).

B.S. Kaliski Jr. et S. Burton : A Layman's Guide to a Subset of ASN.1, BER, and DER. Rapport technique, RSA Laboratories, RSA Data Security Inc., Redwood City, California, USA, novembre 1993. Public-Key Cryptography Standards (PKCS).

G. Karner : Semantic Integration of ASN.1 into SDL. *In SDL'93*, pages 305–315, 1993.

J. Koivisto et J. Reilly : Generating Object-Oriented Telecommunications Software Using ASN.1 Descriptions. *In* G. Neufeld et B. Plattner, éditeurs : *Upper Layer Protocols, Architectures and Applications*, pages 133–147, 1992.

A. J. Kouijzer : Transforming ASN.1 into LOTOS : a User's View, octobre 1990. PTT-Research, Verlag (844/90).

E. Kwast : Automatic ASN.1 Constraint Generation for Testing Purposes. *In Protocol Specification, Testing and Verification (PSTV)*, pages 338–345, Vancouver, Canada, juin 1994.

A. Ladwig et H. König : Datenspezifikation mit ASN.1. *In Nachrichtentechnik Elecktronik*, volume 40 (10), pages 382–385, 1990. En allemand.

John Larmouth : *Understanding OSI*. International Thomson Computer Press, 1996.

R.G. Lavender, D.G. Kafura et R.W. Mullins : Programming with ASN.1 using Polymorphic Types and Type Specialization. *In* M. Medina et N.S. Borenstein, éditeurs : *Upper Layer Protocols, Architectures and Applications*, pages 147–162, 1994.

J.-P. Lemaire : *Les protocoles de réseaux. OSI et DECnet Phase V*. Hermès, 1996. ISBN : 2-86601-559-2.

Xavier Leroy : *The Objective Caml system – Documentation and user's guide*. INRIA-Rocquencourt, Domaine de Voluceau, BP 105, 78153 Le Chesnay Cedex, France, septembre 1997. http ://pauillac.inria.fr/ocaml/.

Huai-An (Paul) Lin : Estimation of the Optimal Performance of ASN.1/BER Transfer Syntax. *In ACM SIGCOMM Computer Communication Review*, volume 23 (3), pages 45–58, juillet 1993.

J. N. MacLeod : Performance of Encoding/Decoding the Transfer Syntax. *In Tenth Annual International Phœnix Conference on Computers and Communications*, pages 644–650, 1991.

Michel Mauny et Daniel de Rauglaudre : Parsers in ml. *In Proceedings of the ACM International Conference on Lisp and Functional Programming*, San Francisco, USA, 1992.

Craig Meyer et G. Chastek : The Use of ASN.1 and XDR for Data Representation in Real-Time Distributed Systems. CMU/SEI-93-TR-10, Carnegie Mellon Univ., Software Engineering Institute, octobre 1993.

N. Mitra : An Introduction to the ASN.1 MACRO Replacement Notation. *AT&T Technical Journal*, pages 66–79, Mai-Juin 1994a.

N. Mitra : Efficient Encoding Rules for ASN.1-Based Protocols. *AT&T Technical Journal*, pages 80–93, Mai-Juin 1994b.

D. Mosberger, L.L. Peterson, P.G. Bridges et S. O'Malley : Analysis of Techniques to Improve Protocol Processing Latency. *In ACM SIGCOMM*, pages 73–84, août 1996.

Gerald W. Neufeld et Son Vuong : An Overview of ASN.1. *In Computer Networks and ISDN Systems*, volume 23, pages 393–415, 1992.

G.W. Neufeld et Y. Yang : The Design and Implementation of an ASN.1-C Compiler. *In IEEE Transaction on Software Engineering*, volume 16 (10), pages 1209–1220, octobre 1990.

Y. Ohara, T. Suganuma et S. Senda : ASN.1 Tools for Semi-automatic Implementation of OSI Application Layer Protocols. *In Second International Symposium on Intyeroperable Information Systems (ISIIS'88)*, Tokyo, Japon, novembre 1988. Elsevier Publishing.

S. O'Malley, T. Proebsting et A.B. Montz : USC : A Universal Stub Compiler. *In ACM SIGCOMM Symposium on Communications Architectures and Protocols*, pages 295–306, Londres, Grande Bretagne, septembre 1994.

C. Partridge et M. T. Rose : A Comparison of External Data Formats. *In* E. Stefferud, O. J. Jacobsen, P. Schicker et Elsevier, éditeurs : *Message Handling Systems and Distributed Applications*, pages 233–245, 1989. (IFIP).

Christian Rinderknecht : Une analyse syntaxique d'ASN.1 en Caml Light. RT-171, INRIA, avril 1995. 238 pages.

Michael Sample : Snacc 1.1 : A High Performance ASN.1 to C/C++ Compiler. Rapport technique, University of British Columbia, Dept. of Computer Science, 6356 Agricultural Rd., Vancouver, British Columbia, Canada, V6T 1Z2, février 1993.

Michael Sample et R. Joop : Snacc 1.2rj : A High Performance ASN.1 to C/C++/IDL Compiler. Rapport technique, University of British Columbia, Dept. of Computer Science, 6356 Agricultural Rd., Vancouver, British Columbia, Canada, V6T 1Z2, 1995.

J. M. Schneider : *Protocol Engineering. A Rule-Based Approach.* Advanced Studies in Computer Science, 1992.

Douglas Steedman : *Abstract Syntax Notation One (ASN.1) - The Tutorial and Reference.* Technology Appraisals, 1990.

Douglas Steedman : Review of Abstract Syntax Notation One. *In* Craig Partridge, éditeur : *ACM Computer Communications Review*, volume 21, avril 1991.

M. Thomas : From 1 Notation to Another One : An ACT-ONE Semantics for ASN.1. *In Second International Conference on Formal Description Techniques for Communication Protocols and Distributed Systems (FORTE)*, pages 517–531, 1990.

M. Thomas : A Translator for ASN.1 into LOTOS. *In FORTE'92*, pages 49–64, octobre 1992.

Pierre Weis et Xavier Leroy : *Le langage Caml*. IIA. InterÉditions, 1993.

J.E. White : ASN.1 and ROS : The Impact of X.400 on OSI. *In IEEE Journal on Selected Areas in Communications*, volume 7, pages 1060–1072. IEEE, septembre 1989.

Claes Wikström : Processing ASN.1 specifications in a declarative language. Rapport technique, Computer Science Laboratory, Ellemtel Utvecklings AB, Box 1505 S-125 25 Älvsjö, Suède. 10 pages.

Table des matières