Sous-Typage par Saturation de Contraintes, Théorie et Implémentation

Benoit Vaugon

NNT : 2016SACLY004

THÈSE DE DOCTORAT
DE
L'UNIVERSITÉ PARIS-SACLAY
PRÉPARÉE À
L'ENSTA-ParisTech

ÉCOLE DOCTORALE N°580
Science et technologie de l'information et de la communication

Spécialité du doctorat : Informatique

Par

M. Benoît Vaugon

Sous-typage par saturation de contraintes
Théorie et implémentation

Thèse présentée et soutenue à Palaiseau, le 15 mars 2016

Composition du jury :

M. Roberto DI COSMO	Professeur, Université Paris-Diderot / INRIA-Paris	Président
M. Hassan AÏT-KACI	Professeur, Université Claude Bernard Lyon 1	Rapporteur
M. François POTTIER	Directeur de recherche, INRIA Paris-Rocquencourt	Rapporteur
Mme Véronique BENZAKEN	Professeur, Université Paris-Sud	Examinatrice
M. Jacques GARRIGUE	Professeur, Université de Nagoya, Japon	Examinateur
M. Manuel SERRANO	Directeur de recherche, INRIA-Méditerranée	Examinateur
M. Michel MAUNY	Professeur, ENSTA-ParisTech	Directeur

Remerciements

Je me rappellerai toujours ma première rencontre avec Michel, alors que j'entrais pour la première fois dans son bureau de l'ENSTA, à Balard. Ses premiers mots furent, alors qu'il se débattait avec son téléphone fixe ne voulant s'arrêter de sonner : « Oui, bonjour Benoît, je suis désolé, mon téléphone devient fou ». J'ai alors découvert quelqu'un de profondément humain, avec qui mes échanges ont toujours été un réel plaisir. Merci Michel pour toutes ces années passionnantes, pour nos longues discussions techniques, en l'air comme je les aime ; mais aussi devant un tableau griffonné en commun à la recherche de théories en premier lieu élégantes, faisant ainsi de la vraie recherche.

Je tiens à remercier tous les membres de mon jury qui ont accepté de passer du temps pour étudier mon travail, et en particulier mes deux rapporteurs François Pottier et Hassan Aït-Kaci, qui m'ont fait, de par leurs visions complémentaires, de nombreuses remarques m'ayant permis d'améliorer mon manuscrit. À ce sujet, merci d'avance au Grand Méchant Loup d'épargner Hassan dans les forêts obscures du sud de Paris.

Merci à François Pessaux sans qui la vie au laboratoire et l'enseignement n'auraient jamais été aussi sympathiques. Merci pour son humour, parfois fin et subtil (je l'entends encore s'exclamer, alors que je lui parlais de mon travail de thèse autour des variants polymorphes : « Tiens ! Des Garriguettes ! ») et parfois beaucoup moins mais toujours aussi hilarant. Merci également à Alexandre, Vladimir et Guillaume qui ont pleinement contribué à rendre fort agréables ces quelques années à l'ENSTA.

Il me faut absolument mentionner ici les petites excursions hebdomadaires qui ont parsemé mon doctorat, au sein d'un regroupement de « spécialistes du chameau ». Que ce soit autour d'une soupe aux tripes, de nouilles étirées ou d'un baby là-haut dans les montagnes, les discussions avec Albin, Benjamin, Çağdaş, Fabrice, Grégoire, Michael, Pierre, Pierrick et Thomas ont toujours été réellement passionnantes. Un merci particulier pour Pierre, toujours prêt à partager ses hacks d'une monstrueuse élégance ; et Benjamin, pour son humour délicat et la qualité inégalable de ses fresques.

Bien avant ma thèse, Emmanuel et Philippe me parlaient déjà de typage et de polymorphisme. Ce sont eux qui, sans nul doute, ont fait naître chez moi l'amour des lettres grecques et des règles d'inférence. Ces dernières années, nos nombreux échanges, généralement vespéraux et autour de burgers parfois atypiques, m'ont beaucoup apporté à la fois scientifiquement et moralement. Je les en remercie avec la plus grande sincérité.

Je tiens à remercier les différents relecteurs de mon manuscrit qui m'ont ainsi permis de l'améliorer : mon père, Emmanuel, Philippe et Gregory.

Je remercie très sincèrement la famille Brouard qui m'a gentiment aidé dans l'organisation de la soutenance.

Merci à tous mes collègues et amis d'Armadillo, avec qui j'ai toujours plaisir à travailler et partager de nombreux moments agréables.

Un grand merci à Marie, Zoé, et Stéphane avec qui les échanges, initialement autour d'une bavette-truffade, se sont transformés au fil du temps en « journées de relativité quantique » et rencontres variées toujours aussi sympathiques. Un merci particulier à Zoé pour tous ces choco-cafés matinaux et discussions spirituellement si instructives.

Merci beaucoup à mes parents, Anne et Marc, mes trois petites cousines, mes oncles et tantes, ma grand-mère, Gwenhadu, Dany, Sandra et Séverine qui m'ont amplement soutenus pendant ces longues années de doctorat.

Enfin, auteure de la mystérieuse phrase : « alors là, c'est le pompon sur le gâteau », Anne-Claire a toujours été là pour moi. Je l'en remercie tendrement.

Introduction

Avant d'exécuter un programme, il est intéressant de vérifier que celui-ci aura un « comportement correct ». Ces vérifications sont bien évidemment nécessaires lorsqu'il s'agit de programmes critiques, c'est-à-dire utilisés dans un contexte où des vies sont en jeu, mais elles sont également bien utiles dans des contextes plus classiques de développement logiciel pour détecter, le plus tôt possible, des problèmes dans un code. Elles permettent ainsi de réduire les temps de développement et de se concentrer sur la logique du code.

Pour ce faire, un principe consiste à écrire des programmes, nommés « analyseurs statiques », servant à vérifier la correction d'autres programmes. Ces analyseurs prennent en entrée du code et cherchent à déterminer s'il est correct ou non vis-à-vis d'une « sémantique ». Ils sont utilisés en général en amont des traducteurs et des interprètes, mais certains peuvent aussi analyser du code généré.

Si cela était possible, il serait intéressant de vérifier que l'exécution d'un programme produira exactement, et dans tous les cas, ce que son auteur — le programmeur — avait prévu. Cela demande néanmoins de décrire, dans un formalisme autre que celui du programme, le comportement attendu. La plupart du temps, ce comportement peut être partiellement décrit grâce à des tests. Bien que très utile, cette approche est rarement suffisante. Lorsqu'un cas a été oublié lors de l'écriture d'un code, il est facilement oublié dans les tests correspondants. Il est également possible de décrire formellement des propriétés du code, typiquement via des formules logiques. Cette approche reste cependant assez rebutante pour le programmeur, mais permet d'approcher une certaine complétude. Malheureusement, la vérification de ces propriétés est souvent longue et difficile, voire indécidable ([S96]).

À défaut de vouloir vérifier la correction complète d'un programme, on se contente en général d'en vérifier un ensemble de « bonnes propriétés ». De base, il est d'usage de vérifier que le texte constituant le programme est correctement structuré, c'est-à-dire qu'il vérifie toutes les conventions de syntaxe du langage dans lequel il est écrit. À l'exception de quelques langages de script, la plupart des langages effectuent ces vérifications avant l'exécution du code.

Au delà de ces vérifications d'ordre « syntaxique », certains environnements de développement, interprètes et compilateurs appliquent des vérifications basiques sur le code grâce à des méthodes ad-hoc. Ces tactiques permettent de contrôler, par exemple, que les variables et fonctions ont bien été définies avant leur utilisation. Pour aller plus loin, certains vérificateurs de ce genre recherchent des « motifs » correspondant à des séquences de code connues comme étant invalides. Bien que simples à mettre en place, ces vérifications ne font que détecter des erreurs classiques des programmeurs et ne prouvent pas la correction du code. Elles sont en général plus restreintes que les analyses que nous allons décrire maintenant.

Une technique classique pour vérifier certaines propriétés des programmes est l'« interprétation abstraite » (cf. [CC77]). Elle est particulièrement bien adaptée pour effectuer des analyses sur des calculs manipulant des valeurs numériques. Cette technique permet par exemple de vérifier que les valeurs de certaines variables entières et flottantes sont toujours comprises dans un certain « domaine », et ainsi de se protéger contre, par exemple, le dépassement des bornes des tableaux et l'envoi d'ordres inadaptés à des actionneurs.

Néanmoins, les propriétés des programmes auxquelles nous nous intéressons dans cette thèse sont très différentes. En particulier, nous ne cherchons jamais ici à représenter des ensembles de valeurs numériques avec une granularité plus fine que « l'ensemble des entiers » ou « l'ensemble des flottants ». À l'inverse, nous nous intéressons à des programmes manipulant des fonctions d'ordre supérieur et des types de données algébriques permettant d'encoder des arbres, des graphes, et des structures de données complexes en général.

Les langages sur lesquels nous travaillons sont des ML (cf. [CD86]) étendus avec en particulier des constructeurs de données et du filtrage de motifs. Les analyses que nous effectuons ici s'inscrivent dans le domaine de l'inférence de types (cf. [PR05]). Nous cherchons ainsi à associer à chaque sous-expression des programmes un « schéma de type » représentant symboliquement l'ensemble des valeurs que l'on est susceptible d'obtenir par évaluation de cette sous-expression.

Ces « schémas de type » sont de la forme « $[\ \forall\alpha_1 \dots \alpha_n\ .\ \alpha_0 \mid \Phi\ ]$ » dans lequel les « variables de type » $\alpha_1, \dots, \alpha_n$ (quantifiées universellement) et α_0 (éventuellement égale à l'un des $\alpha_1, \dots, \alpha_n$), sont des noms symboliques pour des ensembles de valeurs, et Φ est un ensemble de « contraintes » reliées par des opérateurs logiques. Ces opérateurs ne sont dans les premiers chapitres que des conjonctions, mais nous les étendons par la suite avec des disjonctions et des négations. Ces contraintes lient par une relation asymétrique de « sous-typage », notée $(\leqslant)$, les variables de types, à la fois entre elles, mais aussi avec des « types paramétrés » comme `int`, `string`, $\alpha_1 \times \alpha_2$ ou $\alpha_1 \to \alpha_2$ (les types « `int` » et « `string` » étant considérés, par soucis d'homogénéité, comme des types paramétrés à zéro paramètre) et avec d'autres constructions de type que nous présentons par la suite.

Certaines approches plus classiques du typage (cf. [MM82, SP05, PJ06]) manipulent également des ensembles de contraintes, mais dans lesquelles les types sont liés entre eux par une relation d'égalité $(=)$. Toutefois, l'utilisation de relations de sous-typage nous permet de représenter des contraintes plus « précises » (comme « $(\alpha \leqslant \texttt{int})$ » ou « $(\texttt{int} \leqslant \alpha)$ ») que la relation d'égalité avec laquelle il n'est possible de lier α et `int` que par la relation « $(\alpha = \texttt{int})$ ». Notre langage de contraintes à base de relations de sous-typage est donc plus expressif pour définir nos schémas de type que les langages de contraintes ne possédant qu'une relation d'égalité. Il reste toujours possible d'encoder une égalité par une double inégalité.

Cependant, grâce à sa symétrie, la relation d'égalité entre les types est plus « facile » à manipuler, et permet en particulier d'utiliser l'algorithme bien connu « union-find » (cf. [CP15, H91]) pour vérifier la compatibilité des contraintes extraites d'un programme en cherchant une valuation des variables de type. Cet algorithme ne peut malheureusement pas être utilisé pour gérer des relations de sous-typage. Comme nous le verrons dans le chapitre 6, de multiples optimisations sont par

conséquent nécessaires pour conserver des performances raisonnables dans les implémentations de nos systèmes.

Nous montrons donc dans cette thèse qu'il est possible d'utiliser cette expressivité du sous-typage, à la fois d'un point de vue théorique en créant des systèmes de types « puissants » permettant d'inférer des ensembles de valeurs très précis ; mais aussi d'un point de vue pratique en donnant les bases pour implémenter ces systèmes en des typeurs raisonnablement performants.

Dans la conception de nos systèmes de types, un but important que nous cherchons à atteindre est de permettre au maximum l'« inférence », c'est-à-dire la possibilité de calculer automatiquement les schémas de types associés aux différentes sous expressions des programmes sans nécessiter d'annotation de type explicite de la part du programmeur. Bien entendu, lors de la « conception d'un langage de programmation », il peut être raisonnable d'imposer au programmeur d'annoter son code pour améliorer sa lisibilité, même dans des situations dans lesquelles ces annotations sont inutiles pour le typeur. Le fait de pousser au maximum l'« inférabilité » de nos systèmes de types, en plus de son utilité pratique, est principalement motivée par des considérations « conceptuelles » afin d'en améliorer l'« élégance théorique ». Nous avons atteint ce but pour presque toutes les constructions de langage et de typage que nous introduisons dans cette thèse, y compris les types existentiels que nous définissons afin d'encoder une version étendue des « types algébriques gardés » (où « `GADT` »). Seule la « récursion polymorphe » nécessite, chez nous, une annotation de type.

Le formalisme que nous utilisons pour définir nos systèmes consiste à les décrire via uniquement des « règles d'inférence ». Ces règles peuvent en particulier être lues comme les différents composants d'un algorithme d'inférence qui, à partir d'une expression, va tenter de lui calculer un « schéma de type ». Elles peuvent-être classées dans trois catégories :

- Les règles de typage. Il en existe en général une par construction du langage. Dans une vision « algorithmique » du système, elles propagent l'inférence de type aux sous-expressions s'il y en a, et génèrent un ensemble de « contraintes de type ».
- Les règles d'instanciation. Elles spécifient comment « instancier » un schéma de type provenant soit d'une généralisation, soit du typage d'une constante ou d'une primitive polymorphe.
- Les règles de saturation. Elles spécifient comment vérifier que l'ensemble des contraintes accumulées lors du typage sont « valides » et « cohérentes » entre elles.

Notre approche se distingue alors des techniques habituelles sur deux points. En premier lieu, le formalisme que nous utilisons permet de représenter uniformément tout un système de types, au sens où les règles de typage, d'instanciation et de saturation sont utilisées dans le but de construire un unique « arbre d'inférence » représentant à lui seul la « preuve complète de typabilité » de notre programme. Soit cet arbre est constructible, le programme est alors dit « typable » et son exécution se déroulera correctement, soit il existe un noeud de cet arbre pour lequel il est impossible de construire au moins un fils, le programme est alors dit « non-typable » et son exécution est susceptible de partir en erreur. En second lieu, notre technique de vérification de la « cohérence » des contraintes accumulées lors du typage est originale par le fait que nous ne

cherchons volontairement pas à « résoudre » ces contraintes. Ce que l'on entend habituellement par « résoudre » des contraintes dans ce contexte consiste à définir une algèbre de types comme par exemple :

$$\tau ::= \alpha \mid (\tau_1, \ldots, \tau_n)\ \mathtt{t} \mid \top \mid \bot \mid \tau_1 \sqcup \tau_2 \mid \tau_1 \sqcap \tau_2 \mid \ldots$$

et à transformer un ensemble de contraintes d'égalité ou de sous-typage en un τ de cette algèbre. La principale raison de ce choix est l'impossibilité de définir une telle algèbre pour représenter certaines contraintes de type nécessaires dans nos systèmes, en particulier la disjonction de contraintes qui ne peut se représenter ni par une union ($\sqcup$) ni par une intersection ($\sqcap$) dans le cas général. À l'inverse, notre mécanisme de saturation va simplement chercher, comme son nom l'indique, à saturer l'ensemble des contraintes accumulées lors du typage dans le but de traquer des incohérences.

Cette thèse est découpée en six chapitres principaux. Le premier introduit le langage sur lequel nous travaillons et sa sémantique. Ce langage variera très peu au long du manuscrit. Le second définit un système de types de base. Ce système est assez simple mais déjà capable de gérer des variants polymorphes de manière plus puissante que, par exemple, le système de types d'OCaml (cf. [LD14]). Il permet de se familiariser avec le formalisme que nous utilisons pour définir nos règles d'inférence, et d'introduire les techniques de preuve de terminaison et de correction de nos systèmes dans un cadre raisonnable. Les trois chapitres suivants présentent trois extensions orthogonales du système de base. Le premier propose un typage affiné du filtrage, le second étend le mécanisme de généralisation standard de ML (cf. [DM82]), et le troisième introduit une certaine forme de « types existentiels » permettant de définir une version étendue des **GADT** classiques. Chacun de ces chapitres comporte les points principaux des preuves de terminaison et de correction des systèmes qui y sont présentés. Enfin, le dernier chapitre présente des techniques d'implémentation de nos systèmes dans le but d'obtenir des performances de typage correctes, chose nécessaire pour les rendre utilisables en pratique.

Chapitre 1

Langage

Nous présentons ici, sous un angle formel, les choix effectués dans cette thèse concernant le langage étudié et sa sémantique. Ce langage sera commun, à quelques détails près, aux différents systèmes de types que nous présenterons par la suite. Sa sémantique restera également inchangée dans toute cette thèse.

1.1 Formalisation du langage

Le but de cette thèse est d'étudier différentes techniques de sous-typage au dessus d'un langage à la ML (cf. [CD86]). Ces techniques ont toutes pour but d'étendre le nombre de programmes acceptés par le typeur, mais par des biais différents. Nous verrons par la suite qu'en combinant ces approches « orthogonales », nous pouvons obtenir des systèmes de types plus riches permettant de modéliser, sans ajout de construction de langage, des « concepts » de plus haut niveau comme par exemple les « objets ». Pour permettre ces combinaisons, le langage étudié dans cette thèse varie assez peu en fonction des chapitres, tout comme sa sémantique. Il s'agit d'un ML classique étendu avec des « constructeurs de données » et du « filtrage de motifs ».

1.1.1 Le noyau du langage

Le noyau de notre langage est un simple λ-calcul. Il s'agit d'un langage « à expressions », au sens où tout programme dans notre langage est une expression. L'ensemble `e` des expressions est défini par la grammaire suivante :

$$\texttt{e ::= x} \mid \lambda\ \texttt{x . e} \mid \texttt{e}_1\ \texttt{e}_2$$

On nomme « λ-terme » une expression de ce langage. Dans cette définition, comme en λ-calcul classique, `x` représente l'ensemble des variables utilisables dans les programmes, la construction $(\lambda\ \texttt{x . e})$ représente une fonction ayant pour paramètre `x` et pour corps `e`, et la construction $(\texttt{e}_1\ \texttt{e}_2)$ représente l'application de la fonction $\texttt{e}_1$ sur l'argument $\texttt{e}_2$.

Chaque variable apparaissant dans un λ-terme est alors soit libre, soit liée par λ. Les termes auxquels nous nous intéresserons dans cette thèse seront tous « clos » au sens où ils ne contiendront aucune variable libre.

Le λ-calcul est déjà suffisant pour représenter n'importe quel « calcul ». Il est important de remarquer que l'évaluation d'une expression ne se résume pas à une simple réduction des membres du terme un à un jusqu'à obtenir le résultat comme le serait la réduction d'une expression arithmétique classique. Il est en particulier possible d'encoder en un simple λ-terme un calcul qui boucle infiniment. Un exemple très classique d'un tel terme est :

```
(λ x . x x) (λ x . x x)
```

Il est également possible de définir des constantes et des opérateurs sur ces constantes sous forme de λ-termes. Par exemple, il est possible de définir les constantes entières avec la méthode de Church :

```
0 ≜ λ f x . x
1 ≜ λ f x . f x
2 ≜ λ f x . f (f x)
3 ≜ λ f x . f (f (f x))
...
```

et les opérateurs successeur, addition et multiplication ainsi :

```
succ ≜ λ n f x . f (n f x)
 (+) ≜ λ m n . m succ n
 (×) ≜ λ m n f . m (n f)
```

Le lecteur pourra vérifier que ces opérateurs, appliqués à des λ-termes représentant des entiers comme définis précédemment, vérifient bien la sémantique standard des opérateurs arithmétiques. Ce genre de technique peut servir à définir de nombreux concepts calculatoires et structures de contrôle, et c'est ce qui fait la puissance et la beauté du λ-calcul.

Pour l'expressivité du langage, il serait donc suffisant de se limiter au λ-calcul. Néanmoins, dans le cadre plus concret d'un langage de programmation, tout définir à partir de λ-termes est problématique. La syntaxe du langage peut certes masquer la complexité des λ-termes cachés derrière les valeurs manipulées, mais les performances d'un évaluateur de λ-calcul sont rapidement dépassées par celles d'un calculateur travaillant sur des données représentées dans un format « binaire » plus classique.

De plus, du point de vue de l'analyse statique, une représentation de toutes les valeurs sous forme de λ-termes peut affaiblir les vérifications de la cohérence d'un programme. En effet, plusieurs « concepts » différents peuvent être représentés par le même λ-terme et empêcher la détection de certaines confusions dans le code. À l'inverse, il est intéressant d'associer des « types de base » différents aux différentes classes de constantes et d'enrichir le langage en structures de contrôle pour les distinguer lors du typage des programmes.

1.1.2 Les constantes

Comme expliqué précédemment, plutôt que de définir les constantes sous forme de λ-termes, nous préférons étendre la définition des expressions avec un ensemble `c` de « constantes prédéfinies » :

```
e +:= c
c ::= () | true | false | n | s
n ::= 0 | 1 | -1 | ...
s ::= "" | ...
```

où « `+:=` » désigne l'extension d'une règle de grammaire existante avec une ou plusieurs nouvelles constructions de syntaxe.

L'ensemble des constantes n'est volontairement pas figé dans le contexte théorique de cette thèse. Bien évidemment, une implémentation concrète d'un langage de programmation, pour qu'elle soit pratique à utiliser, définira un ensemble de constantes bien fourni. Cet ensemble contiendra typiquement les booléens, les entiers, les flottants, les caractères, les chaînes de caractères, etc.

1.1.3 Les primitives

Notre langage étant muni de constantes définies autrement que par des λ-termes, il est nécessaire de l'enrichir de primitives permettant de manipuler ces constantes. Ces primitives correspondent aux opérateurs de base que l'on trouve classiquement dans les langages : les opérateurs logiques sur les booléens, les opérateurs arithmétiques classiques sur les nombres entiers et flottants, des opérateurs de manipulation des chaînes, etc. Nous enrichissons donc notre langage avec deux constructions syntaxiques correspondant à l'application de primitives unaires (notées $\mathtt{p}^1$) et binaires (notées $\mathtt{p}^2$) :

$$\mathtt{e} \;+{:=}\; \mathtt{p}^1\ \mathtt{e} \mid \mathtt{p}^2\ \mathtt{e}_1\ \mathtt{e}_2$$

$$\mathtt{p}^1 \;::=\; \mathtt{(not)} \mid \mathtt{(\sim -)} \mid \ldots$$

$$\mathtt{p}^2 \;::=\; \mathtt{(+)} \mid \mathtt{(-)} \mid \ldots$$

où l'opérateur $(\sim -)$ est le moins unaire.

Pour des raisons de lisibilité, on s'autorisera par la suite à utiliser les opérateurs unaires en notation préfixe, et les opérateurs binaires en notation infixe avec les règles de priorité standards. On notera ainsi « $\mathtt{e}_1\ \mathtt{+}\ \mathtt{e}_2$ » plutôt que « $\mathtt{(+)}\ \mathtt{e}_1\ \mathtt{e}_2$ ».

Nous remarquerons que notre définition des primitives ne permet de les utiliser dans une expression que en leur passant immédiatement tous leurs arguments. Certains langages comme OCaml permettent d'utiliser les opérateurs seuls en tant que fonctions. Il est ainsi possible d'écrire « `let f = (+) in ...` » ce qui est équivalent à « `let f = (`λ `x y . x + y) in ...` ». Nous préférons ne pas introduire cette notation dans notre définition des expressions car cela compliquerait inutilement les règles de typage et de sémantique. Une telle notation peut alors simplement être vue comme du sucre syntaxique, et donc expansée lors de l'analyse syntaxique des programmes.

Tout comme pour les constantes, notre langage sera donc toujours « paramétré » par un ensemble de primitives. Pour pouvoir travailler avec cet ensemble inconnu de primitives, il devra être fourni avec deux fonctions δ_1 et δ_2 définissant la sémantique des primitives respectivement unaires et binaires (voir la section 1.2), et une fonction T de typage des constantes et des primitives utilisée dans les systèmes de types (voir la section 2.3). Les fonctions T, δ_1 et δ_2 devront également être liées par une relation de compatibilité pour assurer la validité du typage vis-à-vis de la sémantique (voir la section 2.4.5 du chapitre 2).

1.1.4 Les n-uplets

Il est parfois pratique en programmation de regrouper, dans une même valeur, plusieurs valeurs en créant un couple, un triplet, un quadruplet, etc. De telles constructions sont bien sûr encodables avec de simples λ-termes mais pour les mêmes raisons que précédemment, il est en général préférable d'étendre le langage des expressions avec une construction syntaxique spécifique. Pour simplifier les règles de sémantique et de typage, on préférera ne définir que les 2-uplets (ou « couples ») et encoder les n-uplets pour $n \geqslant 3$ comme une imbrication de couples `(`$\mathtt{e}_1$`, (`$\mathtt{e}_2$`, (`$\mathtt{e}_3$`, ...)))`. Nous n'ajoutons alors que la construction de couple à notre définition des expressions :

$$\mathtt{e} \mathrel{+}= \mathtt{(}\mathtt{e}_1\mathtt{,}\ \mathtt{e}_2\mathtt{)}$$

Pour extraire les valeurs d'un couple, l'approche standard consiste à étendre le filtrage de motifs (défini en section 1.1.9). Par simplicité pour les règles de sémantique et de typage, nous préférons réserver, dans cette thèse, le filtrage de motifs aux constructeurs de données (voir section 1.1.5) et définir à la place les deux primitives d'arité 1, `fst` et `snd` dont la sémantique est respectivement d'extraire le premier et le second élément d'un couple :

$$\mathtt{p}^1 \mathrel{+}= \mathtt{fst} \mid \mathtt{snd}$$

1.1.5 Les constructeurs de données

Les constructeurs de données (aussi appelés « variants ») que nous considérons ici permettent d'accoler une « marque » à une valeur. Cette marque est simplement un nom commençant par une majuscule pour le distinguer des noms de variables.

Ils sont très classiques en programmation fonctionnelle et remplacent en une unique construction les « `union` », « `enum` » et « `struct` » de C. Ils permettent d'encoder des structures de données arborescentes, comme par exemple en OCaml des arbres comportant des chaînes aux noeuds et des entiers aux feuilles :

```
type tree = Leaf of int | Node of string * tree * tree
```

Pour des raisons pratiques, les variants prennent dans certains langages un nombre d'arguments variable (aucun, un ou plusieurs). Pour simplifier la sémantique et le typage de notre langage, nous préférons nous limiter ici à des constructeurs de données à un argument, sachant qu'il est

toujours possible de passer un n-uplet en argument pour simuler un constructeur de données à plusieurs arguments, et de passer la constante `()` en argument pour simuler un constructeur de données sans argument. On étend alors la définition des expressions ainsi :

```
e +:= K e
```

Pour simplifier la syntaxe, nous considérerons par la suite qu'un constructeur seul `K` est une expression valide, simple sucre syntaxique pour `K ()`.

Les constructeurs de données étudiés dans cette thèse ont un statut différent en fonction des chapitres. En premier lieu, c'est-à-dire dans le chapitre 3, les constructeurs de données sont similaires aux variants polymorphes d'OCaml. Ils ne sont donc pas « déclarés » ni associés à un « constructeur de type » comme le sont des variants classiques. En revanche, il est possible de les utiliser directement au milieu du code en indiquant simplement le nom du constructeur et son argument.

Le chapitre 4 ne s'intéresse pas aux constructeurs de données et ils y seront donc ignorés par soucis de simplicité.

Dans le chapitre 5, les constructeurs de données étudiés sont une variante des **`GADT`**. Ils sont déclarés dans le code via une construction de syntaxe spécifique leur attribuant des contraintes de sous-typage. La définition exacte des **`GADT`** nécessite quelques prérequis sur le système de types et ne sera donc donnée que lorsque cela sera possible, au début du chapitre 5.

1.1.6 La construction « `let x = e`$_1$ `in e`$_2$ »

Il est courant d'ajouter une construction `let` aux expressions :

```
e +:= let x = e1 in e2
```

La sémantique précise de cette construction est définie dans la section 1.2. Intuitivement, pour évaluer une telle expression en appel par valeur, on commence par évaluer `e`$_1$ et mettre la valeur obtenue « de côté » en l'associant à la variable `x`. On évalue ensuite `e`$_2$ et à chaque fois que l'on y rencontre la variable `x` (et que `x` n'a pas été redéfinie localement par un autre `let x` ou un λ `x`), on l'évalue en la valeur que l'on avait mise de côté comme résultat de l'évaluation de `e`$_1$. La variable `x` peut apparaître plusieurs fois dans `e`$_2$ et la valeur obtenue par évaluation de `e`$_1$ sera alors répliquée autant de fois que nécessaire.

En réalité, la sémantique de cette construction est exactement la même que celle de `((`λ `x . e`$_2$`) e`$_1$`)`. Il y a néanmoins plusieurs intérêts à l'ajouter au λ-calcul. En premier lieu, d'un point de vue « conception de langage », il est en général plus intuitif pour le programmeur d'écrire `(let x = e`$_1$ `in e`$_2$`)` que `((`λ `x . e`$_2$`) e`$_1$`)`. En effet, `e`$_1$ est évaluée avant `e`$_2$ et il est donc plus naturel de l'écrire avant dans le code. De plus, cette construction correspond à la définition d'une « variable locale », commune à la quasi-totalité des langages de programmation.

L'autre intérêt de cette construction concerne le typage à la Damas/Milner (cf. [DM82]). Un traitement particulier, appelé « généralisation » est habituellement effectué en ML sur le typage de

cette construction `let`. Avec cette technique classique de « généralisation », une expression de la forme `(let x = e1 in e2)` est acceptée dans des cas où son équivalent sémantique `((λ x . e2) e1)` ne l'est pas.

Néanmoins, l'intégralité du chapitre 4 est consacré à une autre approche de la généralisation (cf. [DM82]), qui rend en particulier le typage de l'expression `(let x = e1 in e2)` équivalente au typage de `((λ x . e2) e1)`. Dans ce chapitre, l'expression `(let x = e1 in e2)` pourra alors être vue comme du sucre syntaxique sur `((λ x . e2) e1)`.

1.1.7 Le `let` récursif

La construction `let rec` ne sera ajoutée que dans le chapitre 5 concernant les **`GADT`**, car ce n'est que dans ce chapitre qu'elle aura un réel intérêt. Elle sera alors munie d'une annotation de types pour gérer la récursion polymorphe sur un **`GADT`**.

Dans le reste de la thèse, la récursion pourra être encodée simplement via l'utilisation d'un combinateur de point fixe comme :

$$\texttt{fix} \triangleq \lambda\ \texttt{f . ((}\lambda\ \texttt{x . f (}\lambda\ \texttt{v . x x v)) (}\lambda\ \texttt{x . f (}\lambda\ \texttt{v . x x v)))}$$

Nos systèmes de types autorisant les types récursifs, l'utilisation d'un tel λ-terme dans une expression ne posera pas de problème de typage, sauf en cas de récursion polymorphe bien entendu.

1.1.8 La conditionnelle

La conditionnelle comme les autres structures de contrôle peuvent déjà êtres encodées en λ-calcul. Nous préférons ajouter une construction spécifique au langage :

$$\texttt{e +:= if } e_1 \texttt{ then } e_2 \texttt{ else } e_3$$

Ainsi, lorsque l'évaluation de $\texttt{e}_1$ donne la constante `true`, l'évaluation de toute l'expression se réduit à évaluer $\texttt{e}_2$. A contrario, si $\texttt{e}_1$ s'évalue en la constante `false`, seule $\texttt{e}_3$ est évaluée. Si $\texttt{e}_1$ s'évalue en une valeur différente de `true` et de `false`, l'évaluation s'arrête sur une erreur.

Cette construction suppose bien entendu la présence des booléens dans l'ensemble des constantes. La conditionnelle est néanmoins une construction facultative dans un langage muni d'une construction de filtrage de motifs. Il aurait été possible d'encoder les booléens `true` et `false` par les variants `TRUE` et `FALSE`, et de simuler la construction `if-then-else` grâce à un filtrage de motifs défini en section 1.1.9 :

$$\texttt{match e with TRUE} \rightarrow \texttt{e}_1 \parallel \texttt{FALSE} \rightarrow \texttt{e}_2$$

L'ajout de la construction `if-then-else` dans le langage étudié dans cette thèse n'est donc qu'à but pédagogique. Il s'agit en réalité de la seule construction de langage non-triviale permettant de mettre en évidence le fonctionnement des règles de sémantique et de typage.

1.1.9 Le filtrage de motifs

Le « filtrage de motifs » est une construction syntaxique classique permettant de discriminer les valeurs du langage (comme les entiers, les chaînes de caractères, etc.), de « déconstruire » des structures de données (comme les listes, les tableaux, les n-uplets, les enregistrements, les variants, etc.) et d'en extraire les membres s'il y a lieu. Dans cette thèse, pour simplifier, nous ne nous intéresserons qu'à une forme restreinte du filtrage de motifs sur les variants.

Nous étendons donc la définition des expressions avec les deux constructions suivantes :

$$\texttt{e} \mathrel{+}= \texttt{match e with } \texttt{K}_1\ \texttt{x}_1 \to \texttt{e}_1 \parallel \ldots \parallel \texttt{K}_n\ \texttt{x}_n \to \texttt{e}_n$$

$$\texttt{e} \mathrel{+}= \texttt{match e with } \texttt{K}_1\ \texttt{x}_1 \to \texttt{e}_1 \parallel \ldots \parallel \texttt{K}_n\ \texttt{x}_n \to \texttt{e}_n \parallel \texttt{x}_d \to \texttt{e}_d$$

Ces deux constructions comprennent une expression `e` et un ensemble de n « cas » (avec $n \geqslant 1$). Chaque cas est composé d'un motif de la forme $\texttt{K}_i\ \texttt{x}_i$ et d'une expression $\texttt{e}_i$ dans laquelle peut apparaître $\texttt{x}_i$. Tous les $\texttt{K}_i$ d'un même filtrage sont supposés deux à deux distincts, par construction.

La sémantique précise de ces constructions de type `match` est définie dans la section 1.2. Intuitivement, pour évaluer une telle expression, on commence par évaluer l'expression `e`. Si la valeur obtenue est un variant dont le tag est l'un des $\texttt{K}_i$ mentionné dans le motif de l'un des cas, on évalue le $\texttt{e}_i$ correspondant en y associant la variable $\texttt{x}_i$ à l'argument du $\texttt{K}_i$ comme lors de l'évaluation d'un `let`. Sinon, si la construction possède le cas par défaut $\texttt{x}_d \to \texttt{e}_d$, on évalue $\texttt{e}_d$ en y associant $\texttt{x}_d$ à la valeur de `e`, sinon l'évaluation s'arrête sur une erreur.

Pour des raisons de simplicité, nous avons volontairement restreint la construction `match` définie ici au filtrage des variants. Ceci ne limite pas vraiment les possibilités offertes par le langage. En effet, les listes peuvent être encodées grâce à des variants, et le filtrage sur des entiers, chaînes de caractères, etc., peut s'encoder grâce à des `if` imbriqués même s'il est souvent plus lisible d'utiliser un `match` lorsque le nombre de cas est supérieur à 3.

Nous n'avons pas non plus défini le filtrage de motifs imbriqués, c'est-à-dire la possibilité d'écrire des expressions comme :

$$\texttt{match e with } \texttt{K}_1\ (\texttt{K}_2\ \texttt{x}) \to \texttt{e}_2 \parallel \texttt{K}_1\ (\texttt{K}_3\ \texttt{x}) \to \texttt{e}_3 \parallel \texttt{K}_4\ \texttt{y} \to \texttt{e}_4$$

En effet, de tels motifs imbriqués peuvent s'expanser avant le typage en filtrages imbriqués. Pour notre exemple, cela donnerait :

$$\texttt{match e with } \texttt{K}_1\ \texttt{x} \to (\texttt{match x with } \texttt{K}_2\ \texttt{x} \to \texttt{e}_2 \parallel \texttt{K}_3\ \texttt{x} \to \texttt{e}_3) \parallel \texttt{K}_4\ \texttt{y} \to \texttt{e}_4$$

Enfin, nous n'avons pas permis l'utilisation de filtres disjonctifs (aussi nommés « or-patterns ») dans les motifs, c'est-à-dire la possibilité de mentionner, dans un même filtre, la disjonction entre plusieurs constructeurs de données. Avec une telle construction, on pourrait écrire :

$$\texttt{match e with } (\texttt{K}_1\ \texttt{x} \parallel \texttt{K}_2\ \texttt{x}) \to \texttt{e}_{12}$$

signifiant qu'on évalue $\texttt{e}_{12}$ à la fois lorsqu'un $\texttt{K}_1$ ou un $\texttt{K}_2$ est obtenue à l'évaluation de `e`. Les systèmes de types classiques gèrent en général les filtres disjonctifs en interne directement car

ils sont plus subtils à expanser pour obtenir une expression sans filtre disjonctif. Une technique simple d'expansion qui fonctionnerait du point de vue du typage serait de répliquer $\mathtt{e}_{12}$ dans autant de cas que nécessaire ainsi :

$$\mathtt{match\ e\ with\ K_1\ x} \rightarrow \mathtt{e}_{12} \parallel \mathtt{K_2\ x} \rightarrow \mathtt{e}_{12}$$

Cette réplication de $\mathtt{e}_{12}$ risquerait néanmoins de provoquer un ralentissement de la compilation et une expansion du code généré, notamment si plusieurs filtres disjonctifs sont imbriqués les uns dans les autres. Une autre méthode pour expanser les filtres disjonctifs consiste à capturer l'expression commune aux filtres dans une fonction ainsi :

$$\mathtt{let\ f} = \lambda\ \mathtt{x\ .\ e}_{12}\ \mathtt{in}$$

$$\mathtt{match\ e\ with\ K_1\ x} \rightarrow \mathtt{f\ x} \parallel \mathtt{K_2\ x} \rightarrow \mathtt{f\ x}$$

Une telle expansion du filtrage disjonctif provoquerait néanmoins l'allocation implicite d'une fermeture représentant la fonction `f`, et donc un ralentissement dû aux applications de cette fonction. Cette perte de performance peut cependant être compensée par une passe d'optimisation du code après le typage.

1.1.10 La mutabilité

La mutabilité dans un langage peut être exprimée sous différentes formes, par exemple par la donnée de structures mutables comme les tableaux ou les enregistrements mutables. La technique la plus simple pour ajouter de la mutabilité dans un langage à la ML consiste à uniquement ajouter des « références ». Ces « références » représentent des boîtes stockées en mémoire sur lesquelles sont définies trois opérations :

- La création d'une référence sur une autre valeur
- La lecture du contenu d'une référence
- Le remplacement du contenu d'une référence par une autre valeur

Même si la présence de mutabilité dans un langage est parfois très pratique pour encoder certains algorithmes, son ajout complexifie la syntaxe des règles de sémantique et les preuves de validité de manière orthogonale aux travaux présentés dans cette thèse.

Le langage considéré au long de cette thèse sera donc supposé sans structure de données mutables. Nous présenterons néanmoins à la fin du prochain chapitre comment modifier notre système pour gérer la mutabilité. Nous verrons en particulier que les choix faits dans notre approche du typage simplifieront grandement les liens entre mutabilité et variance.

1.1.11 Résumé

Le langage étudié dans cette thèse peut en résumé être défini par la grammaire suivante :

```
e ::=
    | x | λ x . e | e1 e2
    | c
    | p1 e | p2 e1 e2
    | (e1, e2)
    | K e
    | let x = e1 in e2
    | if e1 then e2 else e3
    | match e with K1 x1 → e1 ‖ ... ‖ Kn xn → en
    | match e with K1 x1 → e1 ‖ ... ‖ Kn xn → en ‖ xd → ed
```

avec les ensembles de constantes et de primitives volontairement laissés ouverts :

```
c  ::= () | true | false | 0 | 1 | -1 | ... | "..." | ...
p1 ::= (not) | (~-) | fst | snd | ...
p2 ::= (&&) | (||) | (+) | (-) | ...
```

Ces grammaires définissent l'ensemble des expressions valides dans notre langage. Nous allons, dans la section suivante, définir formellement comment évaluer ces expressions.

1.2 Sémantique

La sémantique consiste à définir précisément la façon dont les expressions sont évaluées, c'est-à-dire la façon dont on associe une valeur à une expression. Il est important de donner une définition formelle de la sémantique de notre langage puisqu'elle sera utilisée pour définir la notion de « validité du système de types » et la démontrer.

La définition que nous donnons ici de l'évaluation d'une expression ne permet pas d'associer une valeur à toutes les expressions. En effet, l'évaluation d'une expression `e` peut dans certains cas ne pas se terminer et « boucler » indéfiniment, et dans d'autres cas se terminer « sur une erreur ».

Il existe différentes manières de définir la sémantique d'un langage. Une méthode classique consiste à définir une sémantique dite « à environnement ». Cette approche consiste à parcourir l'expression en maintenant un « environnement d'évaluation » représentant l'association entre les variables du programme (introduites par un λ, un `let` ou un `match`) et leur valeur. L'avantage principal de cette approche est d'être similaire à la façon mentale d'imaginer l'évaluation d'une expression, ainsi qu'à son implémentation, que ce soit via un interprète ou l'exécution d'un code compilé. Il est même possible de recourir à une représentation des variables sous la forme d'« indices de de Bruijn » pour se rapprocher plus encore du fonctionnement d'un évaluateur à pile.

L'approche que nous choisissons ici est toutefois très différente, il s'agit d'une sémantique dite « à petits pas, par remplacement, et sans environnement », et a été introduite par Wright et Felleisen [WF92] en 1992. Elle s'éloigne de la façon dont les programmes sont habituellement évalués dans un ordinateur et il ne serait en particulier pas raisonnable, pour des questions de performance, d'implémenter un évaluateur de code de cette manière. Cette technique offre néanmoins la même puissance d'expressivité que les autres approches. Son principal avantage est de simplifier les preuves de validité des systèmes de types.

Le principe de base consiste à ne travailler qu'avec en permanence une unique expression « close », représentant l'état courant de l'évaluateur. Cette expression reste « close » au sens où, à chaque étape de l'évaluation, toutes les variables apparaissant dans l'expression restent liées via un λ, un `let` ou un `match`, ou disparaissent.

Une telle sémantique revient à définir une fonction représentant « un pas d'évaluation », associant une expression « close » à une expression « close ». L'évaluation complète d'une expression reviendra donc à la transformer pas à pas pour former une séquence d'expressions, jusqu'à l'obtention d'une expression que l'on ne peut plus évaluer d'un pas, soit parce qu'elle est devenue « invalide », soit parce qu'on a obtenu l'expression représentant la « valeur résultat ».

1.2.1 Les « valeurs »

Pour définir une sémantique à petit pas, il faut commencer par définir l'ensemble des « valeurs ». Cet ensemble, noté `v`, est ici un sous-ensemble de l'ensemble des expressions `e` défini manuellement

par la grammaire suivante :

$\texttt{v} ::= \lambda\ \texttt{x . e} \mid \texttt{c} \mid (\texttt{v}_1, \texttt{v}_2) \mid \texttt{K v}$

Il peut donc s'agir d'une fonction, d'une constante, d'un couple de valeurs, ou d'un constructeur de données ayant comme argument une valeur. Une valeur ne peut donc pas être une variable seule, une primitive, une application, un `let`, un `if-then-else`, un `match-with`, un couple d'expressions qui ne sont pas des valeurs, ni un constructeur de données appliqué à une expression qui n'est pas une valeur. Néanmoins, de telles expressions peuvent apparaître sous le λ d'une valeur.

Par exemple, la constante `3` et le couple `(K "hello", λ x . x + 1)` sont des valeurs, mais l'expression `K (1 + 2)` ne l'est pas.

Nous remarquerons que la grammaire définissant l'ensemble des valeurs est bien compatible avec celle définissant les expressions, et qu'en particulier, toute valeur est, par construction, une expression. En revanche, toutes les expressions ne sont pas des valeurs.

1.2.2 Un « petit pas » pour l'évaluateur

Un « petit pas d'évaluation », noté « $\longrightarrow$ » représente une réduction d'une expression lorsque cela est possible directement via son constructeur de tête. Il ne faut pas le confondre avec ce que nous nommerons par la suite un « grand pas d'évaluation » qui autorisera la réduction d'une sous-expression et sera noté « $\longmapsto$ ».

Un « petit pas d'évaluation » est en réalité une fonction partielle de l'ensemble `e` des expressions dans lui-même. Il est défini par disjonction des cas de la manière suivante :

Lorsque $\delta_1\ (\texttt{p}^1, \texttt{v})$ est défini :	$\texttt{p}^1\ \texttt{v}$	$\longrightarrow$	$\delta_1\ (\texttt{p}^1, \texttt{v})$
Lorsque $\delta_2\ (\texttt{p}^2, \texttt{v}_1, \texttt{v}_2)$ est défini :	$\texttt{p}^2\ \texttt{v}_1\ \texttt{v}_2$	$\longrightarrow$	$\delta_2\ (\texttt{p}^2, \texttt{v}_1, \texttt{v}_2)$
	$(\lambda\ \texttt{x . e})\ \texttt{v}$	$\longrightarrow$	$\texttt{e}[\texttt{x} \mapsto \texttt{v}]$
	$\texttt{let x = v in e}$	$\longrightarrow$	$\texttt{e}[\texttt{x} \mapsto \texttt{v}]$
	$\texttt{if true then e}_1\ \texttt{else e}_2$	$\longrightarrow$	$\texttt{e}_1$
	$\texttt{if false then e}_1\ \texttt{else e}_2$	$\longrightarrow$	$\texttt{e}_2$
	$\texttt{match K v with}\ldots \parallel \texttt{K x} \to \texttt{e} \parallel \ldots$	$\longrightarrow$	$\texttt{e}[\texttt{x} \mapsto \texttt{v}]$
	$\texttt{match K v with}\ldots \parallel \texttt{K x} \to \texttt{e} \parallel \ldots \parallel \texttt{x}_d \to \texttt{e}_d$	$\longrightarrow$	$\texttt{e}[\texttt{x} \mapsto \texttt{v}]$
Lorsque v n'est pas de la forme $\texttt{K}_i\ \texttt{v}$ avec $1 \leqslant i \leqslant n$:			
	$\texttt{match v with K}_1\ \texttt{x}_1 \to \texttt{e}_1 \parallel \ldots \parallel \texttt{K}_n\ \texttt{x}_n \to \texttt{e}_n \parallel \texttt{x}_d \to \texttt{e}_d$	$\longrightarrow$	$\texttt{e}_d[\texttt{x}_d \mapsto \texttt{v}]$

Cette définition entraîne par exemple que, pour toutes expressions $\texttt{e}_1$ et $\texttt{e}_2$, l'image de l'expression $\texttt{(if true then e}_1\ \texttt{else e}_2\texttt{)}$ par la fonction $(\longrightarrow)$ est $\texttt{e}_1$. Autrement dit, il est possible d'effectuer un petit pas d'évaluation sur l'expression $\texttt{if true then e}_1\ \texttt{else e}_2$, et après ce petit pas, nous obtenons l'expression $\texttt{e}_1$.

Cette définition fait intervenir les fonctions partielles δ_1 et δ_2 mentionnées précédemment en 1.1.3. Pour rappel, le langage étudié dans cette thèse est paramétré par un ensemble de constantes et de primitives, qui est fourni avec deux fonctions d'évaluation δ_1 et δ_2 donnant la sémantique de l'application des opérateurs unaires et binaires à des valeurs.

Typiquement, nous aurons :

- $\delta_2\ (\texttt{(+)}, \texttt{1}, \texttt{2}) = \texttt{3}$
- $\delta_1\ (\texttt{not}, \texttt{true}) = \texttt{false}$
- $\delta_1\ (\texttt{fst}, \texttt{(“hello”, “world”)}) = \texttt{“hello”}$
- etc.

La syntaxe `e[x ↦ v]` représente l'expression `e` dans laquelle toutes les occurrences libres de la variable `x` ont été remplacées par la valeur `v`. La valeur `v` peut ainsi disparaître lorsque `x` n'apparaît pas libre dans `e`, et être répliquée lorsque `x` apparaît plusieurs fois libre dans `e`.

Cette fonction $(\longrightarrow)$ n'est pas définie sur les valeurs (ce qui est normal puisque les valeurs sont des expressions déjà évaluées), mais également sur certaines expressions que l'on souhaiterait être capable d'évaluer, comme par exemple :

```
let n = 1 + 2 in n
```

En effet, l'expression « `1 + 2` » n'est pas une valeur et il est impossible d'appliquer la règle « `let x = v in e` $\longrightarrow$ `e[x ↦ v]` » pour la réduire par $(\longrightarrow)$. Pour évaluer de telles expressions, nous allons maintenant définir les notions de « contexte d'évaluation » et de « réduction de sous-expressions ».

1.2.3 Le contexte d'évaluation

La notion de « contexte d'évaluation » ne doit pas être confondue avec la notion d'« environnement d'évaluation » mentionnée précédemment. Il s'agit ici de définir un ensemble `E` des « expressions possédant un trou noté `[]` » et un opérateur, noté « `_[_]` », permettant de remplir ce trou avec une expression standard `e` (ce que l'on notera `E[e]`) pour obtenir une nouvelle expression sans trou. L'ensemble `E` est à nouveau défini par une grammaire :

```
E ::=
    | []
    | E e | v E
    | p¹ E | p² E e | p² v E
    | (E, e) | (v, E) | K E
    | let x = E in e
    | if E then e₁ else e₂
    | match E with K₁ x₁ → e₁ ‖ ... ‖ Kₙ xₙ → eₙ
    | match E with K₁ x₁ → e₁ ‖ ... ‖ Kₙ xₙ → eₙ ‖ x_d → e_d
```

Cette définition nous impose en particulier que tout contexte d'évaluation possède un et seulement un « trou » noté « `[]` ». Ceci nous permet de définir sans ambiguïté l'opérateur (noté « `_[_]` ») prenant en argument un contexte `E`, une expression `e`, et créant l'expression notée `E[e]` dans laquelle le trou `[]` de `E` a été remplacé par `e`.

Par exemple, si le contexte $\texttt{E}_0$ est défini par :

$$\texttt{E}_0 \triangleq \texttt{let x = 1 + [] in x}$$

et l'expression $\texttt{e}_0$ par :

$$\texttt{e}_0 \triangleq \texttt{if true then 3 else 4}$$

alors, l'expression $\texttt{E}_0\texttt{[e}_0\texttt{]}$ vaut :

$$\texttt{E}_0\texttt{[e}_0\texttt{]} = \texttt{let x = 1 + (if true then 3 else 4) in x}$$

Dans la définition de E, le cas $\texttt{p}^2$ `v E` permet de placer un trou dans le deuxième argument de l'application d'un opérateur binaire lorsque le premier argument est déjà évalué (c'est-à-dire est une valeur). C'est ce qui permet d'évaluer le deuxième argument d'un opérateur binaire avant d'exécuter l'opération. Il est important d'imposer que $\texttt{p}^2$ soit un opérateur binaire dans ce cas car sinon, certaines expressions pourraient avoir plusieurs décompositions de la forme `E[e]`. Par exemple, l'expression e définie par :

$$\texttt{e} \triangleq \texttt{fst ((}\lambda\texttt{ x . x + 1), (}\lambda\texttt{ x . x - 1)) (1 + 2)}$$

pourrait se décomposer en $\texttt{E}_1\texttt{[e}_1\texttt{]}$ avec :

$$\texttt{E}_1 = \texttt{[] (1 + 2)} \quad \text{et} \quad \texttt{e}_1 = \texttt{fst ((}\lambda\texttt{ x . x + 1), (}\lambda\texttt{ x . x - 1))}$$

mais aussi en $\texttt{E}_2\texttt{[e}_2\texttt{]}$ avec :

$$\texttt{E}_2 = \texttt{fst ((}\lambda\texttt{ x . x + 1), (}\lambda\texttt{ x . x - 1)) []} \quad \text{et} \quad \texttt{e}_2 = \texttt{1 + 2}$$

Le fait qu'une même expression puisse se décomposer sous la forme `E[e]` de plusieurs manières posera des problèmes d'ambiguïté dans l'ordre d'évaluation. Il est donc nécessaire que la décomposition sous la forme `E[e]` soit unique, lorsqu'elle existe.

À cause de cette contrainte sur la définition de E, un opérateur ne peut pas être à la fois unaire et binaire. Il s'agit d'une hypothèse sur la définition des fonctions δ_1 et δ_2 : les domaines de primitives sur lesquelles elles s'appliquent doivent être disjoints. Dans le contexte d'un compilateur ou d'un évaluateur, l'opérateur arithmétique standard `(-)` est alors problématique, il doit donc être annoté tôt dans la chaîne de compilation comme étant utilisé sous sa forme binaire ou unaire, typiquement au moment de l'analyse syntaxique.

Le contexte d'évaluation va maintenant nous permettre de définir l'évaluation d'une sous-expression, nécessaire à la définition d'un « grand pas d'évaluation ».

1.2.4 Un « grand pas » pour l'évaluation

Un « grand pas d'évaluation », noté par une flèche à talon « $\longmapsto$ », est défini comme le « passage au contexte » d'un petit pas d'évaluation (noté « $\longrightarrow$ »). Tout comme ($\longrightarrow$), il s'agit d'une fonction partielle définie de l'ensemble des expressions `e` dans lui même. Nous commençons par définir la relation ($\longmapsto$) puis démontrons que c'est une fonction.

Par définition, l'expression $\texttt{e}_2$ est image de $\texttt{e}_1$ par la relation ($\longmapsto$) (ce que l'on notera « $\texttt{e}_1 \longmapsto \texttt{e}_2$ ») s'il existe un contexte d'évaluation `E` et deux expressions $\texttt{e}'_1$ et $\texttt{e}'_2$ tels que les trois conditions suivantes soient vérifiées :

- $\texttt{e}_1 = \texttt{E}[\texttt{e}'_1]$
- $\texttt{e}_2 = \texttt{E}[\texttt{e}'_2]$
- $\texttt{e}'_1 \longrightarrow \texttt{e}'_2$

Pour tout contexte `E` et toutes expressions $\texttt{e}'_1$ et $\texttt{e}'_2$, nous avons donc :

$$\texttt{E}[\texttt{e}'_1] \longmapsto \texttt{E}[\texttt{e}'_2] \quad \Longleftrightarrow \quad \texttt{e}'_1 \longrightarrow \texttt{e}'_2$$

Ainsi, à partir de l'expression $\texttt{e}_1$, un grand pas d'évaluation peut être effectué soit directement par un petit pas d'évaluation (dans le cas où `E = []`), soit en évaluant une sous-expression de $\texttt{e}_1$ par un petit pas. La décomposition de $\texttt{e}_1$ sous la forme $\texttt{E}[\texttt{e}'_1]$ étant unique, il n'y a pas le choix dans la sous-expression de $\texttt{e}_1$ à évaluer par un petit pas. Ceci retire toute ambiguïté sur l'ordre d'évaluation des sous expressions. La définition d'un grand pas d'évaluation que nous avons donnée est donc déterministe, toute expression a au plus une image par ($\longmapsto$), ce qui montre que la relation ($\longmapsto$) est bien une fonction.

1.2.5 Les expressions « bloquées »

La fonction ($\longmapsto$) n'est pas définie pour toutes les expressions. Elle n'est en particulier pas définie sur la partie des expressions que sont les valeurs : aucune valeur ne peut subir un grand pas d'évaluation, le calcul est déjà terminé. Il existe néanmoins d'autres expressions que les valeurs sur lesquelles la fonction ($\longmapsto$) n'est pas définie, elles sont nommées les « expressions bloquées ».

Par définition, une « expression bloquée » est une expression $\texttt{e}_0$ qui n'est pas une valeur et telle qu'il n'existe aucune expression $\texttt{e}_1$ vérifiant $\texttt{e}_0 \longmapsto \texttt{e}_1$.

Par exemple, les expressions suivantes sont bloquées :

- `x`
- `3 “hello”`
- `let x = 1 + true in x`

Être une expression bloquée n'est néanmoins pas équivalent au fait de contenir une sous-expression bloquée. Par exemple, les expressions suivantes ne sont pas bloquées :

- `(1 + 2, match 3 with A -> 4)`
- `if false then fst 4 else 7`

1.2.6 Plusieurs pas d'évaluation

Nous allons maintenant définir une relation entre l'ensemble des expressions `e` et lui même, notée « $\longmapsto\!\!\!\!\!\rightarrow$ », représentant « plusieurs pas d'évaluation ». La relation ($\longmapsto\!\!\!\!\!\rightarrow$) est définie comme la fermeture réflexive transitive de ($\longmapsto$) : par définition, $\texttt{e}_1 \longmapsto\!\!\!\!\!\rightarrow \texttt{e}_2$ si l'une des conditions suivantes est vérifiée :

- $\texttt{e}_1 = \texttt{e}_2$
- $\texttt{e}_1 \longmapsto \texttt{e}_2$
- Il existe une expression intermédiaire `e` telle que $\texttt{e}_1 \longmapsto\!\!\!\!\!\rightarrow \texttt{e}$ et $\texttt{e} \longmapsto \texttt{e}_2$

Il s'agit d'une relation et pas d'une fonction car une expression peut avoir plusieurs images par ($\longmapsto\!\!\!\!\!\rightarrow$). Cette relation nous permet alors de définir la fonction d'évaluation.

1.2.7 La fonction d'évaluation

La fonction d'évaluation, notée `eval`, est une fonction totale et non-calculable (au sens de la théorie de la calculabilité (cf. [S96])) sur l'ensemble des expressions `e`. Elle a pour co-domaine l'ensemble ($\texttt{v} \cup \{ \Uparrow, \textsc{error} \}$) et est définie de la manière suivante :

- `eval(e) = v` si $\texttt{e} \longmapsto\!\!\!\!\!\rightarrow \texttt{v}$ avec `v` une valeur
- `eval(e) = ERROR` s'il existe une expression bloquée `e'` telle que $\texttt{e} \longmapsto\!\!\!\!\!\rightarrow \texttt{e'}$
- `eval(e) =` $\Uparrow$ sinon (lorsque l'évaluation boucle)

Comme les valeurs et les expressions bloquées n'ont pas d'image par ($\longmapsto$), une même expression ne peut pas avoir plusieurs images par `eval`. La définition de la fonction `eval` n'est donc pas ambiguë.

Les expressions bloquées jouent donc un rôle très important : ce sont les expressions qui provoqueraient une erreur à l'exécution si on décidait de les interpréter ou de les compiler et d'exécuter le code produit. Détecter qu'une expression est bloquée n'est pas difficile en soi, toute la difficulté du typage tient à prévoir si l'évaluation d'un code aboutira ou pas à une expression bloquée après un nombre indéfini de pas d'évaluation, autrement dit si `eval` renverra ERROR. C'est une de ces techniques de typage que nous étudions dans le chapitre suivant.

Chapitre 2

Système de types de base

Nous présentons ici la technique utilisée pour formaliser les différents systèmes de types à base de saturation de contraintes auxquels nous aurons affaire par la suite. Un système de types de base est ensuite présenté dans ce formalisme, il formera le point de départ des systèmes des prochains chapitres. Nous donnerons ensuite une trame des preuves de validité de tels systèmes de types vis-à-vis de la sémantique du langage, et enfin prouverons la terminaison et la validité du système de base.

2.1 Contexte

Le but du « typage » est principalement d'analyser les programmes dans le but d'y détecter des erreurs. Pour atteindre un tel but, une idée pourrait être d'exécuter effectivement le programme que l'on souhaite analyser et d'observer son comportement. Néanmoins, les programmes ne sont pas toujours déterministes, ils peuvent dépendre d'un certain nombre de facteurs extérieurs sur lesquels nous n'avons pas le contrôle au moment de la vérification du programme comme la lecture d'informations dans un fichier, la réception de données venant d'un capteur, ou l'appel volontaire à une fonction renvoyant une valeur aléatoire. De plus, l'exécution d'un programme peut durer un temps très long, voire ne pas se terminer. Toutefois, les analyses que nous souhaitons faire sur les programmes avant leurs exécutions doivent se terminer en un temps fini, et si possible, en un temps « raisonnable ». Il est donc nécessaire, pour faire de telles analyses, de s'abstraire du comportement exact du programme et de se restreindre à en extraire des « propriétés ».

En réalité, les systèmes de types ont également un autre but que la simple « vérification de validité », à savoir extraire depuis le code des informations utiles pour le programmeur : les « types ». Les types représentent des ensembles de valeurs que peuvent prendre les sous-expressions du programme lors de l'exécution. Ils sont utiles à la fois en terme de documentation, car il est souvent plus facile de lire un type donnant une information synthétique sur un code plutôt que le code lui-même ; mais aussi en terme d'auto-protection de la part du programmeur. En effet, celui-ci peut être amené à annoter son code avec des types (potentiellement incomplets) et demander à l'analyseur (le « typeur », en l'occurrence) de vérifier que ses annotations sont correctes.

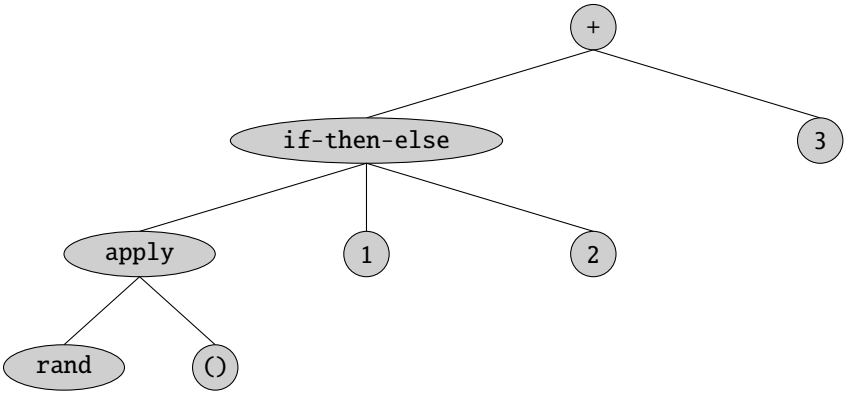
Un système de types a donc pour but de définir formellement quels types sont compatibles avec quelles expressions. Un système de types peut donc être utilisé directement en tant que « validateur » : étant donné une expression et un type, ce type est-il compatible avec cette expression ?

Dans certains cas, il est également possible de transformer un système de types en un algorithme dit d'« inférence » prenant en entrée une expression et générant, s'il y arrive, un type compatible avec cette expression.

Habituellement, la définition d'un système de types se fait indépendamment de la sémantique du langage. Pour que le système de types nous donne une information intéressante vis-à-vis de la sémantique, nous sommes amenés à les relier par un « théorème de validité » : étant donnée une expression `e`, s'il existe un type τ compatible avec `e`, alors l'évaluation de `e` devra se dérouler correctement. Ce théorème est détaillé en section 2.4.3.

2.1.1 Principes de base

Un algorithme de typage possède en général une phase consistant à effectuer un parcours récursif du programme, qui est alors vu comme un arbre, et à en extraire des contraintes. Par exemple, l'expression `((if rand () then 1 else 2) + 3)` sera représenté par l'arbre :



Pour chaque noeud de cet arbre, qui correspond donc à une sous-expression de l'expression originale, nous allons chercher à associer une « variable de type » représentant l'ensemble des valeurs obtenues lors des évaluations de cette sous-expression. Chacune de ces variables de type est contrainte par deux biais : d'une part par la sous-expression en question qui est susceptible de générer un certain ensemble de valeurs à l'exécution, mais aussi par le parent du noeud susceptible d'imposer des contraintes sur les valeurs acceptables comme résultat de l'évaluation de la sous-expression. Nous allons alors construire une conjonction de contraintes (notée « Φ ») contenant l'ensemble des contraintes satisfaites par chacune de ces variables de type.

Par exemple, dans l'arbre précédent, le noeud `if-then-else` s'évaluera parfois en `1` et parfois en `2`, qui sont des entiers, et son parent, le noeud `(+)`, lui impose d'être un entier. Ces deux ensembles étant les mêmes, cette partie du programme devrait s'exécuter sans erreur.

La différence principale entre « unification » et « sous-typage » tient à la forme des contraintes. Un algorithme à base d'unification imposera des contraintes d'égalité entre les types tandis qu'un algorithme à base de sous-typage imposera des contraintes d'inclusion.

Représenter les contraintes par des relations d'inclusion est en général plus « fin » que les représenter par des égalités car il est toujours possible d'encoder une égalité par une double inclusion, tandis que l'inverse est impossible. Les systèmes à base de sous-typage offrent donc plus d'expressivité dans les contraintes reliant les types et sont donc en général plus « puissants » au sens où plus de programmes valides sont acceptés par le typeur.

Néanmoins, les algorithmes à base d'unification peuvent avoir recours à un algorithme de résolution des contraintes d'égalité nommé « union-find » (cf. [CP15, H91]) qui est très performant en pratique, tandis que la saturation de contraintes d'inégalités est souvent plus complexe à implémenter et moins performante.

Malgré cette difficulté implémentatoire, cette thèse se concentre sur le problème du sous-typage. Le chapitre 6 sur l'implémentation donne un ensemble de techniques qui ont été éprouvées pour améliorer les performances d'un typeur à base de sous-typage et rendre la saturation de contraintes utilisable en pratique.

2.1.2 Approches classiques du sous-typage

Il est assez classique de considérer les types comme des structures arborescentes définies par une grammaire ressemblant à :

$$\tau \ ::= \alpha \mid \tau_b \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2$$

$$\tau_b \ ::= \texttt{bool} \mid \texttt{int} \mid \texttt{string} \mid \ldots$$

Dans un système de types à base d'unification, il est ainsi possible d'associer un type seul à une expression, comme par exemple le type « $\alpha \to \alpha \to \texttt{int}$ » dans lequel apparaît la variable de type α libre. Une telle syntaxe est simple et agréable à manipuler pour l'utilisateur.

Dans un monde avec sous-typage, une approche classique consiste à se débarrasser des contraintes de sous-typage en étendant la grammaire des types avec les constructions suivantes :

$$\tau \ +:= \top \mid \bot \mid \tau_1 \sqcup \tau_2 \mid \tau_1 \sqcap \tau_2$$

Grâce à une telle extension, lorsque le langage des contraintes est restreint à une conjonction de contraintes de sous-typage, il est possible de représenter un schéma de type quelconque (possédant des contraintes de sous-typage arbitraire) par un type. Le principe de cette transformation d'un schéma de type en type consiste à réduire la conjonction de contraintes de sous-typage par des règles de la forme :

- $\alpha \leqslant \tau_1 \ \wedge \ \alpha \leqslant \tau_2 \ \rightsquigarrow \ \alpha \leqslant \tau_1 \sqcap \tau_2$
- $\tau_1 \leqslant \alpha \ \wedge \ \tau_2 \leqslant \alpha \ \rightsquigarrow \ \tau_1 \sqcup \tau_2 \leqslant \alpha$
- etc.

Sur cette nouvelle algèbre de types (munie de « $\top$ », « $\bot$ », « $\sqcup$ » et « $\sqcap$ »), il est alors d'usage de définir une « relation de sous-typage », notée également $(\leqslant)$, spécifiant si deux types τ_1 et τ_2

vérifient $\tau_1 \leqslant \tau_2$. Il existe deux approches pour définir cette relation de sous-typage, à savoir le « sous-typage syntaxique » et le « sous-typage sémantique ».

Le sous-typage syntaxique consiste à définir cette relation via un ensemble de règles décomposant les types en fonction de leur forme syntaxique et propageant la relation de sous-typage jusqu'à obtenir des relations élémentaires (comme « `int` $\leqslant$ `int` », « `int` $\leqslant$ `string` » ou « $\top \leqslant \bot$ ») sur lesquelles un ensemble d'axiomes spécifient si la relation de sous-typage est vérifiée ou non.

À l'inverse, le sous-typage sémantique va, en premier lieu, chercher à donner une « sémantique » aux types. Pour ce faire, le principe consiste à définir une « fonction d'interprétation », en général notée $(\llbracket \_ \rrbracket)$, associant un ensemble (une partie d'un certain domaine $\mathcal{D}$) à un type. La relation de sous-typage $(\leqslant)$ entre les types peut alors être définie via la relation d'inclusion sur les ensembles associés :

$$\tau_1 \leqslant \tau_2 \iff \llbracket \tau_1 \rrbracket \subset \llbracket \tau_2 \rrbracket$$

Les travaux de Haruo Hosoya et Benjamin Pierce [HP03, HP01], et en particulier leur travail sur XDuce, s'inscrivent dans la voie du sous-typage sémantique. Une limitation importante de leurs systèmes est l'absence de types fonctionnels. Cette restriction a été levée plus tard par Véronique Benzaken, Giuseppe Castagna et Alain Frisch ([CF05, FC08]) lors de leur travaux liés à $\mathbb{C}$Duce. Ils introduisent entre autres une technique générale permettant de définir la « fonction d'interprétation » sur des constructions de types plus complexes permettant en particulier de gérer le typage des fonctions $(\to)$, des références (`ref`) et de l'évaluation paresseuse (`lazy`).

Les travaux de Pottier en 2001 (cf. [P01]) s'inscrivent quant à eux dans la voie du sous-typage syntaxique en montrant différentes méthodes de simplification des ensembles de contraintes utilisant des mécanismes de résolution.

Les travaux de Gottlieb de 2011 (cf. [G11]) continuent dans cette direction. Ils montrent en particulier comment formaliser et implémenter simplement un système à base de sous-typage sur un langage muni de variants et d'enregistrements.

Stephen Dolan et Alan Mycroft (cf. [DM15]) ont suivi une approche très similaire récemment. En plus de conserver une notion de « type principal » et d'être implémentable, le système qu'ils présentent dans cet article met en évidence certains invariants concernant la forme de leurs contraintes et la position des opérateurs $(\sqcup)$ et $(\sqcap)$ dans leurs types. Ces invariants leur permettent de gagner en élégance en simplifiant leurs règles de typage et l'implémentation associée. Des propriétés très semblables apparaîtront dans nos systèmes. En effet, ces considérations se rapprochent beaucoup de la classification que nous introduirons entre nos types (τ^l et τ^r) et des propriétés de conservation les concernant lors de la saturation.

2.1.3 Manipulation de contraintes sans résolution

Une approche assez différente consiste à uniquement tenter de vérifier la cohérence entre les contraintes extraites lors du typage, sans chercher à les « résoudre ». Une telle approche ne cherche donc pas à faire disparaître les opérateurs logiques entre les contraintes au profit de

constructions de type, et a tendance à enrichir le langage des contraintes plutôt que le langage des types. Cette voie a en particulier été explorée dans les années 1995 par Trifonov et Smith (cf. [ES95, TS96]).

C'est cette direction que nous avons choisie de suivre, en premier lieu pour son élégance (ce qui est très subjectif), mais aussi parce que c'est la seule voie que nous ayons trouvée pour gérer proprement les disjonctions et les négations dans les ensembles de contraintes, et ainsi gagner une expressivité que les mécanismes de résolution nous interdisaient.

2.2 Formalisme utilisé

Le but de cette section est de présenter les mécanismes de base que nous utilisons pour définir les types et les systèmes associés. Les types manipulés dépendent bien entendu du système de types, ils seront donc enrichis au cours des chapitres de cette thèse.

2.2.1 Notre approche

Notre but dans cette thèse est d'étendre les mécanismes de sous-typage. Nous serons alors amenés à étendre le langage des contraintes avec des disjonctions, des négations, et d'autres formes de relations entre les types. Malheureusement, une extension du langage des types avec des unions, des intersections, top et bottom n'est plus suffisante pour représenter de telles contraintes de type. En conséquence, nous n'allons plus chercher à enrichir la grammaire des types pour supprimer les contraintes de sous-typage, mais au contraire, à l'épurer au maximum. À la place, la structure de base qui nous permettra de représenter un « ensemble de valeurs » sera un « schéma de type » et contiendra en particulier un ensemble de contraintes, mais dans un langage beaucoup plus riche.

Il est important de noter que la présence de sous-typage va faire intervenir une relation asymétrique entre les types que nous noterons « $\leqslant$ ». Cette relation représente l'inclusion des ensembles dénotés par les types qu'elle compare. De telles relations sont générées par les systèmes de types que nous allons définir. Cependant, les types qui seront générés à gauche d'un ($\leqslant$) n'ont pas exactement la même structure que ceux qui seront générés à droite. Cette différence sera très légère dans le système de base que nous décrivons dans ce chapitre, mais s'accentuera par la suite. Il serait bien évidemment possible de définir uniquement « l'ensemble de tous les types » comme l'union de l'ensemble des types pouvant apparaître à gauche d'un ($\leqslant$) et de ceux pouvant apparaître à droite. Une telle définition de la structure des types poserait néanmoins quelques problèmes d'élégance, à la fois dans le cadre d'une formalisation car certaines relations seraient syntaxiquement valides mais n'auraient aucun sens et ne pourraient jamais être générées ; mais aussi dans le cadre d'une implémentation car le code du typeur serait alors « pollué » par la gestion de cas impossibles pour des raisons algorithmiques.

Nous sommes donc amenés à définir deux ensembles de types :

$$
\begin{aligned}
\tau^l &\ \texttt{::=}\ \alpha \mid \texttt{(}\alpha_1\texttt{, ..., }\alpha_n\texttt{) t} \mid \texttt{K}\ \alpha \\
\tau^r &\ \texttt{::=}\ \alpha \mid \texttt{(}\alpha_1\texttt{, ..., }\alpha_n\texttt{) t} \\
&\quad \mid \texttt{\{ K}_1\ \alpha_1\ \texttt{‖ ... ‖ K}_n\ \alpha_n\ \texttt{\}} \\
&\quad \mid \texttt{\{ K}_1\ \alpha_1\ \texttt{‖ ... ‖ K}_n\ \alpha_n\ \texttt{‖}\ \alpha_d\ \texttt{\}}
\end{aligned}
$$

L'ensemble τ^l représente les types pouvant apparaître à gauche d'un ($\leqslant$) et τ^r ceux pouvant apparaître à droite.

D'après cette définition, un type est soit :

- une variable de type

- un constructeur de types paramétré par n variables de type (avec $n \geqslant 0$). Nous avons décidé de représenter tous les constructeurs de types de cette manière car ils sont manipulés uniformément dans les règles de saturation. Les constructeurs de types sont utilisés en particulier pour représenter :
  - les types prédéfinis associés aux constantes, typiquement d'arité 0, comme `int`, `bool`, `string`, etc.
  - le type $(\times)$ d'arité 2 utilisé pour les couples. Pour des raisons de lisibilité, on s'autorisera le raccourci de notation « $\alpha_1 \times \alpha_2$ » pour « $(\alpha_1, \alpha_2)(\times)$ »
  - le type $(\rightarrow)$ d'arité 2 utilisé pour les fonctions. De la même manière, on s'autorisera le raccourci de notation « $\alpha_1 \rightarrow \alpha_2$ » signifiant « $(\alpha_1, \alpha_2)(\rightarrow)$ ».
  - les types algébriques gardés (où « **GADT** ») définis par l'utilisateur (voir le chapitre 5).
- dans le cas τ^l, le type d'un variant polymorphe seul provenant de sa construction ; et dans le cas τ^r, un ensemble de variants polymorphes provenant d'une déconstruction via un filtrage de motifs.

Dans cette définition, la seule différence entre τ^l et τ^r concerne donc les types associés aux variants polymorphes, et cela pourrait passer pour un détail. Cependant, nous préférons introduire cette distinction entre τ^l et τ^r dès le départ car elle sera encore plus marquée dans le système de types du chapitre 4 sur la généralisation étendue (τ^l sera alors augmenté de « schémas de type » mais pas τ^r), et primordiale pour démontrer certaines propriétés dans le chapitre 5 sur les **GADT**.

En réalité, la distinction entre τ^l et τ^r est beaucoup plus profonde qu'une simple différence syntaxique. Ils sont chacun associé à un ensemble bien distinct de constructions du langage :

- Un type τ^l est toujours associé à la « construction » d'une valeur. Il en sera généré lors du typage des constantes, des couples, des fonctions, etc.
- Un type τ^r est quant à lui associé à la « déconstruction » d'une valeur. Il en sera généré lors du typage de l'application (impliquant la déconstruction de la fonction), de la conditionnelle (qui déconstruit le booléen associé au test), du filtrage de motifs, etc.

À différentes occasions, nous serons amenés à « renommer » les variables présentes dans les types. Pour ce faire, nous utiliserons la notation « $\tau[\alpha_1 \mapsto \alpha_2]$ » représentant le type τ dans lequel toutes les occurrences libres de α_1 ont été remplacées par α_2. Un renommage de variables peut être injectif (au sens où les variables ont des images deux à deux distinctes) ou non. Par défaut, dans ce manuscrit, le terme « renommage » désignera un renommage potentiellement non-injectif.

Une propriété notable des mécanismes de saturation de contraintes que nous présentons ici est que les types ne changent jamais de côté dans les comparaisons. Ainsi, si un type construit (c'est-à-dire autre qu'une variable de type) a été généré en tant que τ^l, il ne deviendra jamais un τ^r. De plus, la seule raison de « clash » de typage est lorsqu'un type construit τ^l est comparé avec un type construit τ^r incompatible. Autrement dit, un clash de typage ne peut apparaître que lorsque la construction d'une valeur se collisionne avec une déconstruction incompatible, par exemple lorsqu'une valeur est construite comme un couple puis utilisée en tant que fonction dans une application.

Cette propriété est très utile pour générer de « bons » messages d'erreur de sous-typage. En effet, il est alors possible d'associer à tout clash une position dans le code indiquant la construction d'une valeur, et une autre position indiquant l'endroit où elle est déconstruite de manière incompatible. Une version avancée peut même utiliser la chaîne de variables de type reliant ces deux types incompatibles, et la présenter au programmeur comme un « chemin d'exécution » invalide du programme.

Une telle propriété n'est néanmoins valide que dans un système à base de sous-typage. Un système à base d'unification ne vérifie pas cette propriété et des clash peuvent apparaître entre deux τ^l ou entre deux τ^r. Par exemple, l'expression (`if ... then 3 else "hello"`) sera typiquement rejetée par un système à base d'unification avec un clash `int`/`string` alors qu'il n'existe aucun chemin d'exécution reliant un `int` à une `string`.

Une autre différence importante entre notre définition de τ^l/τ^r et l'approche standard tient au fait que notre définition des types n'est pas récursive. Une telle définition a l'avantage de forcer les types et les schémas de type à rester sous une « forme normale ». De plus, cette représentation des types trivialise les preuves de terminaison des algorithmes de saturation car il suffit de borner le nombre de variables de type pour borner le nombre de types, et ainsi le nombre de contraintes de types. Cependant, cette définition nous empêche d'imbriquer des types les uns dans les autres, et donc en particulier d'écrire un type comme « $\alpha \to \alpha \to$ `int` ». Nous aurons donc besoin d'un « schéma de type » pour dénoter un tel ensemble.

2.2.2 Schémas de type

Dans ce chapitre, les contraintes de types que nous allons manipuler sont uniquement des relations de sous-typage. Elles seront enrichies par la suite avec d'autres sortes de relations entre les types. L'ensemble de ces contraintes, noté `C`, est donc défini pour l'instant par la grammaire suivante :

$$\texttt{C ::= } \tau^l \leqslant \tau^r$$

Pour le système de types de base, un ensemble de contraintes, noté Φ, se résume à une conjonction de `C` :

$$\Phi \texttt{ ::= \{ C}_1 \texttt{ } \wedge \texttt{ ... } \wedge \texttt{ C}_n \texttt{ \}}$$

La définition de Φ sera enrichie par la suite avec des disjonctions et des négations. Les Φ que nous manipulerons à partir de maintenant seront considérés comme des « ensembles » de contraintes dans lesquels l'ordre des relations dans la conjonction n'a pas d'importance. On s'autorisera en particulier les notations « $C \in \Phi$ » signifiant que l'un des membres de la conjonction Φ est égal à C, et « $\Phi_1 \subset \Phi_2$ » signifiant que tous les C de Φ_1 appartiennent à Φ_2.

Un schéma de type, quant à lui, est défini de la manière suivante :

$$\sigma \texttt{ ::= [} \forall\alpha_1 \texttt{ ... } \alpha_n \texttt{ . } \alpha \texttt{ | } \Phi \texttt{]}$$

Il contient :

- Un ensemble de n variables de type $\alpha_1 \ldots \alpha_n$: les variables « généralisées ».
- Une variable de type (appartenant à $\alpha_1 \ldots \alpha_n$ ou pas) : la racine du schéma.
- Un ensemble de contraintes Φ liant α avec les α_i, potentiellement d'autres variables de type et éventuellement des constructeurs de types et des variants.

Par exemple, l'ensemble des fonctions ayant deux paramètres d'un même type quelconque et renvoyant un entier est habituellement noté $(\alpha \to \alpha \to \texttt{int})$. Il sera ici représenté par le schéma de type :

$$\sigma = [\ \forall \alpha_1 \alpha_2 \alpha_3\ .\ \alpha \mid \{\ \alpha_1 \to \alpha_2 \leqslant \alpha \wedge \alpha_1 \to \alpha_3 \leqslant \alpha_2 \wedge \texttt{int} \leqslant \alpha_3\ \}\ ]$$

Un tel schéma de type est certes assez peu lisible. Son avantage est qu'il est basé sur des briques élémentaires très simples, ce qui facilite sa manipulation dans le système de types et dans les preuves. Pour être utilisable en pratique, une implémentation réelle d'un système de types basé sur une telle représentation devra définir une fonction d'affichage mettant les schémas de type sous une forme plus classique lorsqu'ils doivent être lus par un programmeur. Une telle technique est détaillée dans le chapitre 6 sur l'implémentation.

2.2.3 Environnement de typage

Le typage d'une expression se fait par un parcours récursif de l'arbre la représentant. Lors de la descente du typeur à travers un `let`, un λ ou un cas de filtrage, une nouvelle variable du programme est créée et peut alors apparaître dans le corps de la construction en question. Le typeur va donc être amené à stocker des contraintes de types concernant ces variables du programme dans une structure appelée « environnement de typage » et notée Γ. De telles contraintes sont simplement représentées par un schéma de type. L'ensemble Γ est donc défini par une liste de couples :

$$\Gamma\ ::= (\mathtt{x}_1,\ \sigma_1),\ \ldots,\ (\mathtt{x}_n,\ \sigma_n)$$

Nous définissons alors deux opérateurs sur Γ :

- L'opérateur d'ajout d'une nouvelle association $(\mathtt{x},\ \sigma)$ dans Γ, noté : « $\Gamma \oplus (\mathtt{x},\ \sigma)$ »
- L'accès à une variable `x` dans un environnement Γ, noté : « $\Gamma[\mathtt{x}]$ »

Lorsqu'un même `x` est lié à plusieurs σ dans Γ, la syntaxe $\Gamma[\mathtt{x}]$ donne accès au dernier σ ajouté pour `x`.

2.2.4 Règles d'inférence

Nous allons maintenant définir comment nous formalisons nos systèmes de types dans cette thèse. Les choix que nous faisons ici sont guidés par un but assez pragmatique : nous souhaitons que nos règles d'inférence soient transformables, de manière systématique, en un algorithme d'inférence.

Le but de l'inférence est, lorsque c'est possible, d'associer à une expression e un schéma de type σ représentant une approximation de l'ensemble des valeurs obtenues par évaluation de e. Chaque système de types que nous allons définir consistera en un ensemble de règles d'inférence. Il en existe trois catégories :

- Les « règles de typage », nommées «T...», de la forme :

$$\frac{\begin{array}{c}\text{T}\ldots\\ \ldots \quad \ldots \quad \ldots\end{array}}{\Phi, \Gamma \vdash \mathsf{e} : \alpha \rhd \Phi'}$$

  De telles règles se lisent de la manière suivante : étant donné un ensemble de contraintes Φ, dans l'environnement de typage Γ, pour que l'expression e soit de type α, Φ doit être enrichi avec de nouvelles contraintes pour obtenir Φ'.

- Les « règles de saturation », nommées «S...», de la forme :

$$\frac{\begin{array}{c}\text{S}\ldots\\ \ldots \quad \ldots \quad \ldots\end{array}}{\Phi \vdash \tau^l \leqslant \tau^r \rhd \Phi'} \qquad \text{et} \qquad \frac{\begin{array}{c}\text{S}\ldots\\ \ldots \quad \ldots \quad \ldots\end{array}}{\Phi \vdash \tau^l \leq \tau^r \rhd \Phi'}$$

  De telles règles se lisent : étant donné un ensemble de contraintes Φ, l'ajout de la contrainte de sous-typage $\tau^l \leqslant \tau^r$ nécessite d'enrichir Φ avec de nouvelles contraintes pour générer Φ'.

- Les « règles d'instanciation » de la forme :

$$\frac{\begin{array}{c}\text{I}\ldots\\ \ldots \quad \ldots \quad \ldots\end{array}}{\Phi \vdash \sigma \leqslant \tau^r \rhd \Phi'}$$

  Ces règles sont similaires aux règles de saturation, sauf qu'elles comparent un schéma de type avec un type plutôt que deux types entre eux.

L'algorithme d'inférence fonctionne de la manière suivante : à partir d'une expression e, nous générons une variable de type α et cherchons à construire un arbre d'inférence dont les noeuds sont des instances de nos règles de typage, de saturation et d'instanciation :

$$\dfrac{\dfrac{\ldots \quad \ldots}{\ldots} \quad \dfrac{\ldots \quad \ldots \quad \ldots}{\ldots} \qquad \dfrac{\dfrac{\ldots \quad \ldots}{\ldots} \quad \dfrac{\ldots \quad \ldots}{\ldots}}{\ldots} \qquad \dfrac{\dfrac{\ldots \quad \ldots}{\ldots} \quad \dfrac{\ldots \quad \ldots}{\ldots}}{\ldots}}{\varnothing, \varnothing \vdash \mathsf{e} : \alpha \rhd \Phi}$$

Lorsque la construction de cet arbre est impossible, l'expression e est dite « non-typable ». On dit alors qu'il se produit un « clash » de typage. Lorsque cet arbre est constructible, il représente une « preuve de typage » de notre expression e qui est alors dite typable. Nous extrayons alors de

cet arbre l'ensemble de contraintes de sous-typage Φ et l'ensemble $\{ \alpha_1, \ldots, \alpha_n \}$ des variables de type générées au cours de l'inférence. Le schéma de type inféré pour e représentant l'ensemble des valeurs possibles par évaluation e est alors :

$$\sigma = [\ \forall \alpha_1 \ \ldots \ \alpha_n \ .\ \alpha \mid \Phi\ ]$$

Le schéma de type σ généré par cet algorithme contient souvent des contraintes qui ne sont pas liées à α. Pour des raisons de performance, il peut être intéressant de le nettoyer et de le normaliser. Ce genre d'algorithme est détaillé dans le chapitre 6 sur l'implémentation.

La syntaxe de nos règles de typage se distingue des systèmes de types « standard » par le fait qu'elle impose de lier l'expression e à une variable de type et pas à un type comme habituellement. Toutes les contraintes de type associées à e sont en réalité regroupées dans Φ'. Il serait bien entendu possible d'autoriser la présence d'un τ^r à la place de α dans les règles de typage, mais ceci ne ferait que compliquer la forme des règles et n'apporterait aucune expressivité supplémentaire pour l'écriture de systèmes de types.

Nous remarquerons la présence de deux formes de règles de saturation différentes, chacune ayant en conclusion à une relation de sous-typage distincte : « $\leqslant$ » et « $\leq$ ». Il s'agit en réalité d'une simple astuce technique permettant de formaliser le « calcul de point fixe » effectué lors de la saturation en utilisant uniquement des « règles ». Il n'y a pas de « différence sémantique » entre ces deux relations. L'usage de deux relations permet simplement d'éviter de boucler lors de la saturation.

Le principe de la saturation est simple : lorsqu'une règle de typage, d'instanciation ou de saturation engendre une contrainte de sous-typage $(\tau_1 \leqslant \tau_2)$, si cette contrainte est déjà présente dans l'ensemble de contraintes Φ, on ne fait rien. Sinon, on l'ajoute à Φ et on génère une contrainte de la forme $(\tau_1 \leq \tau_2)$ qui est prise en charge par les autres règles de saturation.

2.3 Règles du système de types de base

Nous définissons ici un système de types de base pour notre langage. Il se compose de règles de typage, de règles de saturation et d'une règle d'instanciation. Ce système peut déjà servir à l'implémentation d'un typeur fournissant du sous-typage classique sur un langage à la ML. Dans le contexte de cette thèse, son principal but est de mettre en évidence les principes de fonctionnement de nos systèmes de types. Ce système sera enrichi dans les chapitres à venir, à la fois par l'ajout et la transformation de certaines règles, mais aussi par des modifications de la structure même des règles.

2.3.1 Règle d'instanciation

Nous commençons par définir une règle d'instanciation qui sera utilisée pour le typage des constantes, des variables, et de l'application des primitives :

$$\textsc{Inst}\quad \frac{\texttt{let}\ \alpha'_1,\ldots,\alpha'_n\ \textit{fresh} \quad \Phi \vdash \alpha_0[\alpha_i \mapsto \alpha'_i]_{i=1}^n \leqslant \tau^r \rhd \Phi_1 \qquad \Phi_1 \vdash \mathsf{C}_1[\alpha_i \mapsto \alpha'_i]_{i=1}^n \rhd \Phi_2 \quad \cdots \quad \Phi_p \vdash \mathsf{C}_p[\alpha_i \mapsto \alpha'_i]_{i=1}^n \rhd \Phi_{p+1}}{\Phi \vdash [\ \forall\alpha_1\ \ldots\ \alpha_n\ .\ \alpha_0 \mid \mathsf{C}_1 \wedge \cdots \wedge \mathsf{C}_p\ ] \leqslant \tau^r \rhd \Phi_{p+1}}$$

Cette règle prend en entrée un ensemble de contraintes Φ, un schéma de type $[\ \forall\alpha_1\ \ldots\ \alpha_n\ .\ \alpha_0 \mid \mathsf{C}_1 \wedge \cdots \wedge \mathsf{C}_p\ ]$ et un type τ^r ; et construit un nouvel ensemble de contraintes Φ_{p+1} dans lequel nous avons ajouté la contrainte $(\alpha_0[\alpha_i \mapsto \alpha'_i]_{i=1}^n \leqslant \tau^r)$ ainsi que les p contraintes $\mathsf{C}_1[\alpha_i \mapsto \alpha'_i]_{i=1}^n$, …, $\mathsf{C}_p[\alpha_i \mapsto \alpha'_i]_{i=1}^n$. La notation $X[\alpha_i \mapsto \alpha'_i]_{i=1}^n$ signifie que l'on remplace dans X toutes les occurrences libres de tous les α_i par leur α'_i correspondant.

Algorithmiquement parlant, pour comparer un schéma de type $[\ \forall\alpha_1\ \ldots\ \alpha_n\ .\ \alpha_0 \mid \Phi_0\ ]$ avec un type τ^r, on doit générer n variables de type $\alpha'_1,\ldots,\alpha'_n$ fraîches. S'il existe un i_0 tel que $\alpha_0 = \alpha_{i_0}$, cela signifie que la variable α_0 est « généralisée » dans le schéma de type. On renomme alors α_0 en α_{i_0} et enrichissons Φ avec la contrainte $\alpha'_{i_0} \leqslant \tau^r$. Sinon, α_0 n'est pas renommée et on enrichit Φ avec la contrainte $\alpha_0 \leqslant \tau^r$. D'autre part, on enrichit Φ avec toutes les contraintes de Φ_0 provenant du schéma de types en y remplaçant toutes les occurrences des variables généralisées α_i par leurs α'_i correspondant.

2.3.2 Règles de typage

Les règles de typage que nous définissons sont dirigées par la syntaxe du langage. Ainsi, étant donnée une expression, c'est la nature du constructeur de tête de cette expression (c'est-à-dire s'il s'agit d'un `if`, d'un `let`, d'une constante, etc.) qui va déterminer la règle de typage à appliquer. Il existe donc une règle de typage par construction du langage. La partie de l'arbre d'inférence contenant des règles de typage sera alors structurellement isomorphe à l'arbre représentant l'expression à typer. Algorithmiquement parlant, le typage d'une expression reviendra donc bien à un parcours récursif de l'arbre la représentant.

Constantes et primitives

Certaines constantes et primitives de notre langage sont « polymorphes ». Leur typage nécessite une fonction que nous noterons T leur associant à chacun un schéma de type qui sera instancié à chacune de leur occurrence. Par exemple, nous aurons :

- $T(\texttt{42}) = [\ \forall\alpha_0\ .\ \alpha_0 \mid \texttt{int} \leqslant \alpha_0\ ]$
- $T(\texttt{false}) = [\ \forall\alpha_0\ .\ \alpha_0 \mid \texttt{bool} \leqslant \alpha_0\ ]$
- $T(\texttt{[]}) = [\ \forall\alpha_0\,\alpha_1\ .\ \alpha_0 \mid \alpha_1\ \texttt{list} \leqslant \alpha_0\ ]$ (où `[]` représente la liste vide)
- $T(\texttt{not}) = [\ \forall\alpha_0\,\alpha_1\,\alpha_2\ .\ \alpha_0 \mid \alpha_1 \leqslant \texttt{bool} \wedge \texttt{bool} \leqslant \alpha_2 \wedge \alpha_1 \to \alpha_2 \leqslant \alpha_0\ ]$

Le typage des constantes se fait alors simplement grâce à la règle suivante :

$$\frac{\Phi \vdash T(\mathtt{c}) \leqslant \alpha \rhd \Phi'}{\Phi, \Gamma \vdash \mathtt{c} : \alpha \rhd \Phi'}\ \textsc{TConst}$$

Pour que la constante `c` soit de type α, on impose que $T(\texttt{c})$ soit plus petit ou égal à α. Puisque $T(\texttt{c})$ est un schéma de type, la prémisse de cette règle est de la forme de la conclusion de la règle Inst. Dans l'arbre d'inférence, tout noeud instance de TConst aura donc pour fils un noeud instance de Inst.

Le typage de l'application d'une primitive nécessite un peu plus de travail :

$$\frac{\texttt{let}\ \alpha_1, \alpha_2\ \textit{fresh} \qquad \Phi \vdash T(\mathtt{p}^1) \leqslant \alpha_1 \to \alpha_2 \rhd \Phi' \qquad \Phi' \vdash \alpha_2 \leqslant \alpha \rhd \Phi'' \qquad \Phi'', \Gamma \vdash \mathtt{e}_1 : \alpha_1 \rhd \Phi'''}{\Phi, \Gamma \vdash \mathtt{p}^1\ \mathtt{e}_1 : \alpha \rhd \Phi'''}\ \textsc{TApplyPrim1}$$

Ainsi, pour inférer les contraintes de sous-typage sur α de telle sorte que l'application de la primitive $\mathtt{p}^1$ à l'expression $\mathtt{e}_1$ soit de type α, on commence par générer deux variables fraîche α_1 et α_2. La primitive $\mathtt{p}^1$ doit en particulier être une fonction. On impose alors que le schéma de type $T(\mathtt{p}^1)$ soit plus petit que $\alpha_1 \to \alpha_2$ (ce qui entraîne l'utilisation directe de la règle Inst). Par ailleurs, on relie α_2 et α par la relation de sous typage ($\alpha_2 \leqslant \alpha$), et on impose que l'argument $\mathtt{e}_1$ soit de type α_1.

La génération d'une seconde variable de type peut surprendre. Cette règle pourrait en effet s'écrire de manière presque équivalente comme suit :

$$\frac{\texttt{let}\ \alpha_1\ \textit{fresh} \qquad \Phi \vdash T(\mathtt{p}^1) \leqslant \alpha_1 \to \alpha \rhd \Phi' \qquad \Phi', \Gamma \vdash \mathtt{e}_1 : \alpha_1 \rhd \Phi''}{\Phi, \Gamma \vdash \mathtt{p}^1\ \mathtt{e}_1 : \alpha \rhd \Phi''}\ \textsc{TApplyPrim1(fake)}$$

Néanmoins, définir la règle TApplyPrim1 de la sorte casserait un invariant important pour la suite : on souhaite que lors de l'inférence des contraintes de sous-typage générées par le typage de $(e : \alpha)$, toutes les contraintes imposées sur α soient de la forme $(\tau^l \leqslant \alpha)$. Introduire le α comme paramètre

d'un type peut générer, via les règles de saturation que nous allons définir, une contrainte de sous-typage de la forme $(\alpha \leqslant \tau^r)$. En dehors du fait que contraindre α en lui imposant d'être sous-type d'un autre type n'aurait aucun sens dans ce contexte, l'introduction d'une telle contrainte pourrait faire réduire la puissance du système de types (typiquement en cas d'ajout de mutabilité dans le langage), et compliquerait certaines preuves de validité (en particulier celle du chapitre 4). Nous utiliserons donc par la suite la première définition de TApplyPrim1. Cette remarque est également valable pour les règles TApplyPrim2 et TApp que nous allons définir.

Le typage de l'application d'une primitive d'arité 2 est très similaire. Pour rappel, les types manipulés par notre système sont en forme normale. Il est impossible d'imbriquer des types les uns dans les autres. Nous utilisons donc une variable intermédiaire α_0 pour représenter le type de la primitive $\mathtt{p}^2$ qui prend deux arguments :

$$\textsc{TApplyPrim2}\quad \frac{\begin{array}{l}\texttt{let}\ \alpha_0, \alpha_1, \alpha_2, \alpha_3\ \textit{fresh} \qquad \Phi \vdash \alpha_0 \leqslant \alpha_2 \to \alpha_3 \rhd \Phi' \qquad \Phi' \vdash \alpha_3 \leqslant \alpha \rhd \Phi'' \\ \Phi'' \vdash T(\mathtt{p}^2) \leqslant \alpha_1 \to \alpha_0 \rhd \Phi''' \qquad \Phi''', \Gamma \vdash \mathtt{e}_1 : \alpha_1 \rhd \Phi'''' \qquad \Phi'''', \Gamma \vdash \mathtt{e}_2 : \alpha_2 \rhd \Phi'''''\end{array}}{\Phi, \Gamma \vdash \mathtt{p}^2\ \mathtt{e}_1\ \mathtt{e}_2 : \alpha \rhd \Phi'''''}$$

Variables

Les variables sont introduites dans notre langage par les constructions λ, `let`, et `match`. Comme nous le verrons par la suite, le typage de ces constructions enrichit l'environnement de typage Γ en y ajoutant l'association entre chaque variable et un schéma de type transportant les contraintes concernant cette variable. Le typage des variables du programme se fait grâce à la règle suivante :

$$\textsc{TVar}\quad \frac{\texttt{when}\ \Gamma[\mathtt{x}]\ \textit{defined} \qquad \Phi \vdash \Gamma[\mathtt{x}] \leqslant \alpha \rhd \Phi'}{\Phi, \Gamma \vdash \mathtt{x} : \alpha \rhd \Phi'}$$

Il consiste simplement à instancier le schéma qui lui est associé dans Γ et à le comparer au type attendu α. Si la variable `x` n'apparaît pas dans Γ, cela signifie qu'elle n'a pas été définie précédemment par un λ, un `let` ou un `match` et le typage s'arrête sur une erreur.

Fonctions et applications

Le typage de la création d'une fonction est défini par la règle suivante :

$$\textsc{TLambda}\quad \frac{\texttt{let}\ \alpha_1, \alpha_2\ \textit{fresh} \qquad \Phi, \Gamma \oplus (\mathtt{x}, \alpha_1) \vdash \mathtt{e} : \alpha_2 \rhd \Phi' \qquad \Phi' \vdash \alpha_1 \to \alpha_2 \leqslant \alpha \rhd \Phi''}{\Phi, \Gamma \vdash \lambda\ \mathtt{x}\ .\ \mathtt{e} : \alpha \rhd \Phi''}$$

Nous commençons par générer deux variables de type α_1 et α_2. Nous typons alors le corps de la fonction avec α_2 en enrichissant l'environnement de typage Γ avec l'association (x, α_1) (où α_1 est

un abus d'écriture pour désigner le schéma de type s'instanciant toujours en $\alpha_1 : [\ \forall\ .\ \alpha_1 \mid \varnothing\ ]$). Enfin, nous imposons que $\alpha_1 \to \alpha_2$ soit sous-type du type attendu pour la fonction, c'est-à-dire α. Remarquons en particulier que cette construction de syntaxe $\lambda \mathtt{x}\ .\ \mathtt{e}$ permet de « construire » une fonction. Elle génère donc une flèche $(\alpha_1 \to \alpha_2)$ apparaissant à gauche d'un $(\leqslant)$.

Le typage d'une application, quant à lui, « déconstruit » une fonction, et donc génère une flèche à droite d'un $(\leqslant)$:

$$\textsc{TApp}\quad \frac{\texttt{let}\ \alpha_1, \alpha_2, \alpha_3\ \textit{fresh} \qquad \Phi \vdash \alpha_1 \leqslant \alpha_2 \to \alpha_3 \rhd \Phi' \qquad \Phi' \vdash \alpha_3 \leqslant \alpha \rhd \Phi'' \qquad \Phi'', \Gamma \vdash \mathtt{e}_1 : \alpha_1 \rhd \Phi''' \qquad \Phi''', \Gamma \vdash \mathtt{e}_2 : \alpha_2 \rhd \Phi''''}{\Phi, \Gamma \vdash \mathtt{e}_1\ \mathtt{e}_2 : \alpha \rhd \Phi''''}$$

Comme le montre la règle précédente, pour typer une application ($\mathtt{e}_1\ \ \mathtt{e}_2$) avec le type attendu α, nous commençons par générer trois variables de type fraîches α_1, α_2 et α_3 ; et imposons qu'elles vérifient $(\alpha_1 \leqslant \alpha_2 \to \alpha_3)$ et $(\alpha_3 \leqslant \alpha)$. Puis, nous typons $\mathtt{e}_1$ avec α_1 et $\mathtt{e}_2$ avec α_2. Cette règle génère bien une flèche à droite d'un $(\leqslant)$.

Tout comme pour la règle TApplyPrim1, nous avons préféré générer une variable de type intermédiaire α_3 plutôt que d'utiliser directement α.

Couples et constructeurs de données

Le typage des couples et des constructeurs de données est très similaire :

$$\textsc{TPair}\quad \frac{\texttt{let}\ \alpha_1, \alpha_2\ \textit{fresh} \qquad \Phi, \Gamma \vdash \mathtt{e}_1 : \alpha_1 \rhd \Phi' \qquad \Phi', \Gamma \vdash \mathtt{e}_2 : \alpha_2 \rhd \Phi'' \qquad \Phi'' \vdash \alpha_1 \times \alpha_2 \leqslant \alpha \rhd \Phi'''}{\Phi, \Gamma \vdash (\mathtt{e}_1,\ \mathtt{e}_2) : \alpha \rhd \Phi'''}$$

$$\textsc{TConstr}\quad \frac{\texttt{let}\ \alpha'\ \textit{fresh} \qquad \Phi, \Gamma \vdash \mathtt{e} : \alpha' \rhd \Phi' \qquad \Phi' \vdash \mathtt{K}\ \alpha' \leqslant \alpha \rhd \Phi''}{\Phi, \Gamma \vdash \mathtt{K}\ \mathtt{e} : \alpha \rhd \Phi''}$$

Nous commençons par typer le ou les argument(s) puis construisons le τ^l correspondant et imposons qu'il soit sous-type du type attendu α. Ces deux constructions « construisent » des valeurs, ce qui explique pourquoi elles génèrent des contraintes de types ayant un type construit à gauche. Les déconstructions correspondantes (à savoir les projections `fst` et `snd`, et le filtrage de motifs) génèrent quant à elles des types construits à droite.

Conditionnelle

Le typage de la conditionnelle se fait grâce à la règle suivante :

$$\textsc{TIf}\quad \frac{\texttt{let}\ \alpha'\ \textit{fresh} \qquad \Phi \vdash \alpha' \leqslant \texttt{bool} \rhd \Phi' \qquad \Phi', \Gamma \vdash \texttt{e}_1 : \alpha' \rhd \Phi'' \qquad \Phi'', \Gamma \vdash \texttt{e}_2 : \alpha \rhd \Phi''' \qquad \Phi''', \Gamma \vdash \texttt{e}_3 : \alpha \rhd \Phi''''}{\Phi, \Gamma \vdash \texttt{if}\ \texttt{e}_1\ \texttt{then}\ \texttt{e}_2\ \texttt{else}\ \texttt{e}_3 : \alpha \rhd \Phi''''}$$

Remarquons qu'il s'agit de la seule règle qui aurait été plus courte si on avait autorisé un τ^r dans les conclusions des règles de typage plutôt qu'imposer une variable de type. À cause de cette contrainte, nous sommes obligés de générer une variable de type intermédiaire $\alpha_1 \leqslant \texttt{bool}$. C'est un faible prix à payer pour éviter de polluer toutes les règles avec des τ^r. Remarquons qu'une implémentation efficace de cette règle pourra toujours réutiliser la même variable de type α_1.

Typage du « `let` »

La règle de typage du `let` est la plus compliquée conceptuellement :

$$\textsc{TLet}\quad \frac{\texttt{let}\ \alpha'\ \textit{fresh} \qquad \Phi, \Gamma \vdash \texttt{e}_1 : \alpha' \rhd \Phi' \qquad \Phi', \Gamma \oplus (\texttt{x}, \textsc{gen}(\alpha', \Phi', \Gamma)) \vdash \texttt{e}_2 : \alpha \rhd \Phi''}{\Phi, \Gamma \vdash \texttt{let}\ \texttt{x} = \texttt{e}_1\ \texttt{in}\ \texttt{e}_2 : \alpha \rhd \Phi''}$$

Elle fait intervenir une fonction de généralisation notée GEN créant un schéma de type à partir d'une variable de type, d'un ensemble de contraintes Φ et d'un environnement de typage Γ. Une définition naïve (mais néanmoins valide et non-restrictive) de cette fonction de généralisation est la suivante :

$$\textsc{gen}(\alpha, \Phi, \Gamma) \triangleq [\ \forall(\textsc{ftv}(\Phi) \backslash \textsc{ftv}(\Gamma))\ .\ \alpha \mid \Phi\ ]$$

où $\textsc{ftv}(\Phi)$ et $\textsc{ftv}(\Gamma)$ représentent respectivement l'ensemble des variables de type libres de Φ et de Γ. Cette définition de GEN est dite « naïve » puisqu'elle génère un schéma de type contenant des contraintes de type (et a fortiori des variables de type) qui ne sont pas liées à α et qui ne seront par conséquent pas utilisées après instanciation. Il peut être intéressant de le nettoyer pour gagner en mémoire, en vitesse et en nombre de variables de type générées. Une version déjà nettement plus performante consiste à ne prendre de Φ que les contraintes de types liées directement ou indirectement à α. De telles optimisations sont décrites dans le chapitre 6 sur l'implémentation.

Filtrage de motifs

Le filtrage de motifs correspond à deux constructions de langage (avec et sans cas par défaut). Nous définissons alors une règle de typage pour chacune de ces constructions :

TMatch

$$\frac{\begin{array}{c}\texttt{let}\ \alpha_e, \alpha_1, \ldots, \alpha_n\ \textit{fresh} \\ \Phi \vdash \alpha_e \leqslant \{\, \mathtt{K}_1\ \alpha_1 \parallel \ldots \parallel \mathtt{K}_n\ \alpha_n \,\} \rhd \Phi_0 \qquad \Phi_0, \Gamma \vdash \mathtt{e} : \alpha_e \rhd \Phi_1 \\ \Phi_1, \Gamma \oplus (\mathtt{x}_1,\ \alpha_1) \vdash \mathtt{e}_1 : \alpha \rhd \Phi_2 \qquad \cdots \qquad \Phi_n, \Gamma \oplus (\mathtt{x}_n,\ \alpha_n) \vdash \mathtt{e}_n : \alpha \rhd \Phi_{n+1}\end{array}}{\Phi, \Gamma \vdash \mathtt{match}\ \mathtt{e}\ \mathtt{with}\ \mathtt{K}_1\ \mathtt{x}_1 \to \mathtt{e}_1 \parallel \ldots \parallel \mathtt{K}_n\ \mathtt{x}_n \to \mathtt{e}_n : \alpha \rhd \Phi_{n+1}}$$

TMatchDefault

$$\frac{\begin{array}{c}\texttt{let}\ \alpha_e, \alpha_1, \ldots, \alpha_n, \alpha_d\ \textit{fresh} \\ \Phi \vdash \alpha_e \leqslant \{\, \mathtt{K}_1\ \alpha_1 \parallel \ldots \parallel \mathtt{K}_n\ \alpha_n \parallel \alpha_d \,\} \rhd \Phi_0 \qquad \Phi_0, \Gamma \vdash \mathtt{e} : \alpha_e \rhd \Phi_1 \\ \Phi_1, \Gamma \oplus (\mathtt{x}_1,\ \alpha_1) \vdash \mathtt{e}_1 : \alpha \rhd \Phi_2 \qquad \cdots \qquad \Phi_n, \Gamma \oplus (\mathtt{x}_n,\ \alpha_n) \vdash \mathtt{e}_n : \alpha \rhd \Phi_{n+1} \\ \Phi_{n+1}, \Gamma \oplus (\mathtt{x}_d,\ \alpha_d) \vdash \mathtt{e}_d : \alpha \rhd \Phi_{n+2}\end{array}}{\Phi, \Gamma \vdash \mathtt{match}\ \mathtt{e}\ \mathtt{with}\ \mathtt{K}_1\ \mathtt{x}_1 \to \mathtt{e}_1 \parallel \ldots \parallel \mathtt{K}_n\ \mathtt{x}_n \to \mathtt{e}_n \parallel \mathtt{x}_d \to \mathtt{e}_d : \alpha \rhd \Phi_{n+2}}$$

Chacune de ces règles, correspondant à la « déconstruction » d'un variant, entraîne la création d'un τ^r. Le typage du filtrage est en fait assez similaire au typage de la conditionnelle, à la nuance qu'il y a n cas au lieu de deux. La variable de type fraîche utilisée pour typer l'expression $\mathtt{e}$ sur laquelle on filtre est simplement comparée avec le τ^r contenant les informations sur les différents variants, ainsi que la variable de type utilisée pour le typage du cas par défaut s'il y a lieu.

2.3.3 Règles de saturation

Certaines des règles de typage et d'instanciation décrites précédemment ont en prémisse des constructions de la forme « $\Phi \vdash \tau^l \leqslant \tau^r \rhd \Phi'$ ». Ces constructions signifient qu'on souhaite ajouter la (potentiellement) nouvelle contrainte de typage $\tau^l \leqslant \tau^r$ dans Φ et vérifier qu'elle est « compatible » avec les contraintes déjà accumulées dans Φ.

La « compatibilité » d'un ensemble de contraintes revient ici à vérifier qu'il ne contient pas de chaîne $\tau^l \leqslant \alpha_1 \leqslant \alpha_2 \leqslant \cdots \leqslant \alpha_n \leqslant \tau^r$ où τ^l et τ^r sont des types construits (c'est-à-dire autres que des variables de type) incompatibles.

Pour vérifier la compatibilité entre un ensemble de contraintes (dont la compatibilité a déjà été prouvée) et une nouvelle contrainte, les règles de saturation vont, comme leur nom l'indique, tenter de « saturer » l'ensemble de contraintes Φ selon la règle de transitivité suivante (naturelle en théorie des ensembles) :

$$\tau^l \leqslant \alpha \ \wedge\ \alpha \leqslant \tau^r \ \Rightarrow\ \tau^l \leqslant \tau^r$$

jusqu'à l'obtention d'un point fixe.

Si une incompatibilité est rencontrée lors de la saturation, une erreur de typage est levée. Si le point fixe est atteint, les contraintes sont prouvées compatibles et le typage continue avec le nouvel ensemble de contraintes saturé.

Règles de point fixe

Nous définissons la recherche du point fixe grâce à deux règles.

- Si une nouvelle contrainte $\tau^l \leqslant \tau^r$ apparaît, on l'ajoute dans Φ et on impose de démontrer que $\tau^l \leq \tau^r$ est compatible avec les contraintes déjà présentes :

$$\textsc{SNewConstraint}\quad \frac{\texttt{when}\ (\tau^l \leqslant \tau^r) \notin \Phi \qquad \Phi \wedge \tau^l \leqslant \tau^r \vdash \tau^l \leq \tau^r \rhd \Phi'}{\Phi \vdash \tau^l \leqslant \tau^r \rhd \Phi'}$$

- Si une contrainte est déjà présente dans Φ, c'est qu'elle a déjà été prouvée compatible avec les autres, ou qu'elle est en train d'être prouvée compatible (ce qui arrive en présence de types cycliques). Dans les deux cas, on laisse Φ inchangé et il n'y a rien de plus à démontrer.

$$\textsc{SAlreadyProved}\quad \frac{\texttt{when}\ (\tau^l \leqslant \tau^r) \in \Phi}{\Phi \vdash \tau^l \leqslant \tau^r \rhd \Phi}$$

Comparaisons entre constructeurs de types

Dans le système simple que nous considérons ici, nous nous limiterons (comme quasiment tous les langages typés classiques) à imposer que deux constructeurs de types sont compatibles s'ils sont de même nom. Dans un système plus permissif, on pourrait imaginer autoriser certaines relations entre constructeurs de types comme par exemple `int` $\leqslant$ `float`. Pour ce faire, il suffirait d'ajouter une règle de typage spécifique, et de mettre à jour la sémantique du langage. Quoi qu'il en soit, lorsqu'une contrainte de sous-typage apparaît entre deux constructeurs de type de même nom, la contrainte de sous-typage est alors reportée sur leurs arguments. L'arité des constructeurs de types étant imposée par leur nom (les types de base (`int`, `float`, etc.) sont d'arité 0 ; la flèche et le produit cartésien sont d'arité 2, etc.), deux constructeurs de types de même nom ont toujours le même nombre d'arguments. La règle suivante formalise ce comportement :

$$\textsc{STypeConstr}\quad \frac{\begin{array}{c}\Phi_1 \vdash \alpha_1 \leqslant \alpha'_1 \rhd \Phi'_1 \qquad \Phi'_1 \vdash \alpha'_1 \leqslant \alpha_1 \rhd \Phi_2 \\ \cdots \\ \Phi_n \vdash \alpha_n \leqslant \alpha'_n \rhd \Phi'_n \qquad \Phi'_n \vdash \alpha'_n \leqslant \alpha_n \rhd \Phi_{n+1}\end{array}}{\Phi_1 \vdash (\alpha_1, \ldots, \alpha_n)\ t \leq (\alpha'_1, \ldots, \alpha'_n)\ t \rhd \Phi_{n+1}}$$

Cette règle de saturation est en particulier valable à la fois pour le produit cartésiens ($\times$) et pour la flèche ($\rightarrow$). La variance des paramètres des constructeurs de types est volontairement ignorée, tous

les constructeurs de types sont considérés invariants sur tous leurs arguments. Comme nous le verrons par la suite, ceci n'entraîne aucune limitation au niveau du sous-typage. En particulier, le typage nous autorise à passer en argument une valeur de type $\{ A \mid B \}$ à une fonction acceptant un argument de type $\{ A \mid B \mid C \}$. Ceci est dû au fait que la variance de la flèche n'est pas encodée dans notre système au niveau des règles de saturation, mais au niveau des règles de typage TLambda et TApp.

Le fait que les types n'aient pas besoin de transporter d'information de variance a deux impacts majeurs :

- L'ajout d'un système de modules munis de « types abstraits » est plus simple, aucune information de variance n'est à ajouter sur les types abstraits. La variance des paramètres de type sera simplement encodée dans les types des valeurs qui les manipulent et qui sont exportées par le module.
- L'ajout de mutabilité dans notre langage ne demandera aucune modification des règles de saturation, la variance nécessaire à la manipulation de données mutables sera simplement encodée dans les schémas de type associés aux primitives d'accès aux données mutables.

Sous-typage sur les variants

La règle de typage TConstr génère des τ^l de la forme $\mathtt{K}\ \alpha$ et les règles TMatch et TMatchDefault génèrent des τ^r de la forme $\{ K_1\ \alpha_1 \parallel \ldots \parallel K_n\ \alpha_n \}$ et $\{ K_1\ \alpha_1 \parallel \ldots \parallel K_n\ \alpha_n \parallel \alpha_d \}$. Lorsque de tels τ^l et τ^r se rencontrent, ils sont gérés par les règles de saturation suivantes :

- Lorsque le constructeur de données $\mathtt{K}$ est l'un des $\mathtt{K}_i$, la comparaison est reportée sur leur argument :

$$\textsc{SVariantMatch}\ \frac{\Phi \vdash \alpha \leqslant \alpha' \rhd \Phi'}{\Phi \vdash \mathtt{K}\ \alpha \leq \{ \ldots \parallel \mathtt{K}\ \alpha' \parallel \ldots \} \rhd \Phi'}$$

- Lorsque le constructeur de données $\mathtt{K}$ n'apparaît pas parmi les $\mathtt{K}_i$ mais que le cas par défaut est présent, la comparaison est reportée sur ce dernier :

$$\textsc{SVariantDefault}\ \frac{\texttt{when}\ \forall i \in [\,1;\ n\,] \mid \mathtt{K} \neq \mathtt{K}_i \qquad \Phi \vdash \mathtt{K}\ \alpha \leqslant \alpha_d \rhd \Phi'}{\Phi \vdash \mathtt{K}\ \alpha \leq \{ \mathtt{K}_1\ \alpha_1 \parallel \ldots \parallel \mathtt{K}_n\ \alpha_n \parallel \alpha_d \} \rhd \Phi'}$$

- Lorsque ce n'est pas un variant qui est comparé mais un constructeur de types, le cas par défaut doit être présent et la comparaison est reportée sur lui :

$$\textsc{SConstrDefault}\ \frac{\Phi \vdash (\alpha_1, \ldots, \alpha_p)\ t \leqslant \alpha'_d \rhd \Phi'}{\Phi \vdash (\alpha_1, \ldots, \alpha_p)\ t \leq \{ \mathtt{K}_1\ \alpha'_1 \parallel \ldots \parallel \mathtt{K}_n\ \alpha'_n \parallel \alpha'_d \} \rhd \Phi'}$$

Cette dernière règle autorise l'exécution d'un filtrage sur une valeur qui n'est pas un variant et suppose que l'évaluation passera alors par le cas par défaut, ce qui est bien compatible avec la sémantique définie précédemment. Ce comportement est moins classique dans les langages de programmation. Il s'agit d'une version généralisée des types « nullables » comme présentés dans l'article [**nullables**]. Il est ainsi possible de représenter la valeur « `NULL` » par un variant, de la « mélanger » avec les autres valeurs du langage et d'utiliser le filtrage pour la différencier, le tout de manière sûre. Cette technique ne se restreint pas à `NULL`, elle permet d'encoder des sémantiques intéressantes mélangeant plusieurs valeurs similaires à `NULL` (comme « `UNDEFINED` », « `UNKNOWN` », etc.) que l'on rencontre dans différents langages, souvent non-typés ou typés dynamiquement.

Pour que l'implémentation d'un tel langage fonctionne, la représentation dynamique des valeurs doit néanmoins nous permettre de différencier, à l'exécution, un variant de toute autre valeur du langage. Il est alors impossible de représenter les variants constants par des entiers comme c'est le cas en OCaml. En revanche, une représentation efficace des variants constants peut se faire par des adresses non-alignées en mémoire, ou par des valeurs allouées à l'initialisation du programme.

Transitivité

La saturation de Φ doit également « court-circuiter » les chaînes de la forme $\tau^l \leqslant \alpha_1 \leqslant \alpha_2 \leqslant \cdots \leqslant \alpha_n \leqslant \tau^r$ présentes dans Φ et ainsi mener à comparer les τ^l et τ^r provenant des extrémités de ces chaînes. Pour se faire, nous enrichissons les règles de saturation de la manière suivante :

- Une variable de type est toujours inférieure ou égale à elle même :

$$\textsc{SSameVar}\quad \frac{}{\Phi \vdash \alpha \leq \alpha \rhd \Phi}$$

- Lorsqu'un type τ^l, qui n'est pas une variable de type ($\tau^l \notin \{\ \alpha\ \}$), est un sous-type de α, on extrait tous les sur-types τ^r de α connus dans Φ, et on leur impose d'être sur-type de τ^l. Il existe plusieurs manières d'exprimer cette propriété par des règles de saturation. Une manière assez "pure" serait de l'exprimer par les deux règles suivantes :

$$\textsc{STransRight}\quad \frac{\begin{array}{l}\texttt{when } \tau^l \notin \{\ \alpha\ \} \ \wedge\ (\alpha \leqslant \tau^r) \in \Phi \ \wedge\ (\tau^l \leqslant \tau^r) \notin \Phi \\ \Phi \wedge \tau^l \leqslant \tau^r \vdash \tau^l \leq \alpha \rhd \Phi' \qquad \Phi' \vdash \tau^l \leq \tau^r \rhd \Phi''\end{array}}{\Phi \vdash \tau^l \leq \alpha \rhd \Phi''}$$

$$\textsc{STransRightEnd}\quad \frac{\texttt{when } \tau^l \notin \{\ \alpha\ \} \ \wedge\ \forall \tau^r\ .\ (\alpha \leqslant \tau^r) \in \Phi \ \Rightarrow\ (\tau^l \leqslant \tau^r) \in \Phi}{\Phi \vdash \tau^l \leq \alpha \rhd \Phi}$$

  Néanmoins, cette représentation, en plus de s'éloigner de l'implémentation (constituée simplement de l'application d'un `List.fold_left` bien choisi), complique les preuves de terminaison et de validité car elle est en partie redondante avec les règles SNewConstraint et

SAlreadyProved. Nous préférons plutôt définir une fonction :

$$\textsc{rights}(\alpha,\ \Phi) \triangleq \{\ \tau^r \mid (\alpha \leqslant \tau^r) \in \Phi\ \}$$

et la règle :

$$\frac{\begin{array}{l}\textsc{STransRight}\\ \texttt{when}\ \tau^l \notin \{\ \alpha\ \} \qquad \texttt{let}\ \tau_1^r, \ldots, \tau_n^r = \textsc{rights}(\alpha,\ \Phi_1)\\ \Phi_1 \vdash \tau^l \leqslant \tau_1^r \rhd \Phi_2 \qquad \cdots \qquad \Phi_n \vdash \tau^l \leqslant \tau_n^r \rhd \Phi_{n+1}\end{array}}{\Phi_1 \vdash \tau^l \leq \alpha \rhd \Phi_{n+1}}$$

qui est beaucoup plus simple à manipuler dans les preuves.

- De manière similaire, lorsque la contrainte à prouver est de la forme $\alpha \leqslant \tau^r$, nous définissons la fonction :

  $$\textsc{lefts}(\alpha,\ \Phi) \triangleq \{\ \tau^l \mid (\tau^l \leqslant \alpha) \in \Phi\ \}$$

  et la règle correspondante :

  $$\frac{\begin{array}{l}\textsc{STransLeft}\\ \texttt{when}\ \tau^r \notin \{\ \alpha\ \} \qquad \texttt{let}\ \tau_1^l, \ldots, \tau_n^l = \textsc{lefts}(\alpha,\ \Phi_1)\\ \Phi_1 \vdash \tau_1^l \leqslant \tau^r \rhd \Phi_2 \qquad \cdots \qquad \Phi_n \vdash \tau_n^l \leqslant \tau^r \rhd \Phi_{n+1}\end{array}}{\Phi_1 \vdash \alpha \leq \tau^r \rhd \Phi_{n+1}}$$

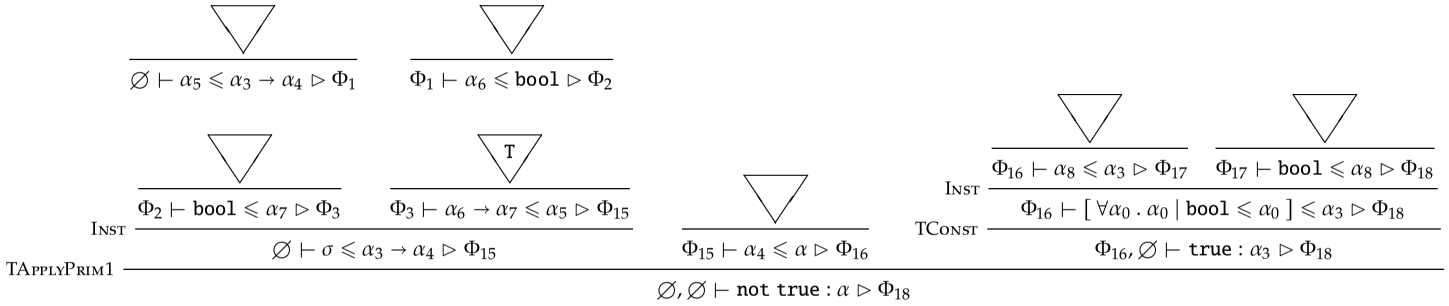
- Lorsqu'on compare deux variables de type, il faut et il suffit de transférer la comparaison sur les sous-types de la première avec les sur-types de la deuxième :

  $$\frac{\begin{array}{l}\textsc{STransLeftRight}\\ \texttt{when}\ \alpha_1 \neq \alpha_2 \qquad \texttt{let}\ \tau_1^l, \ldots, \tau_n^l = \textsc{lefts}(\alpha_1,\ \Phi_{1,1}) \qquad \texttt{let}\ \tau_1^r, \ldots, \tau_p^r = \textsc{rights}(\alpha_2,\ \Phi_{1,1})\end{array}}{\begin{array}{ccc}\Phi_{1,1} \vdash \tau_1^l \leqslant \tau_1^r \rhd \Phi_{1,2} & \cdots & \Phi_{1,p} \vdash \tau_1^l \leqslant \tau_p^r \rhd \Phi_{2,1}\\ \vdots & & \vdots\\ \Phi_{n,1} \vdash \tau_n^l \leqslant \tau_1^r \rhd \Phi_{n,2} & \cdots & \Phi_{n,p} \vdash \tau_n^l \leqslant \tau_p^r \rhd \Phi_{n+1,1}\end{array}}{\Phi_{1,1} \vdash \alpha_1 \leq \alpha_2 \rhd \Phi_{n+1,1}}$$

La saturation de Φ vis-à-vis des règles de transitivité peut bien entendu être implémentée de manière beaucoup plus performante qu'en parcourant à chaque nouvelle comparaison de la forme $\tau^l \leqslant \alpha$ l'ensemble de contraintes Φ en quête des sur-types de α. Une représentation adéquate de l'ensemble Φ permettra en particulier d'accéder en temps constant à tous les sur-types et tous les sous-types de toute variable de type.

2.3.4 Exemple

Pour illustrer l'utilisation de ces règles, donnons par exemple la partie typage de l'arbre obtenu pour le typage de l'expression « `not true` » :



avec :

- $\sigma = [\ \forall \alpha_0\ \alpha_1\ \alpha_2\ .\ \alpha_0 \mid \alpha_1 \leqslant \texttt{bool} \wedge \texttt{bool} \leqslant \alpha_2 \wedge \alpha_1 \to \alpha_2 \leqslant \alpha_0\ ]$

- $\Phi_1 = \alpha_5 \leqslant \alpha_3 \to \alpha_4$
- $\Phi_2 = \Phi_1 \wedge \alpha_6 \leqslant \texttt{bool}$
- $\Phi_3 = \Phi_2 \wedge \texttt{bool} \leqslant \alpha_7$
- $\Phi_4 = \Phi_3 \wedge \alpha_6 \to \alpha_7 \leqslant \alpha_5$
- $\Phi_5 = \Phi_4 \wedge \alpha_6 \to \alpha_7 \leqslant \alpha_3 \to \alpha_4$
- $\Phi_6 = \Phi_5 \wedge \alpha_6 \leqslant \alpha_3$
- $\Phi_7 = \Phi_6 \wedge \alpha_3 \leqslant \alpha_6$
- $\Phi_8 = \Phi_7 \wedge \alpha_3 \leqslant \texttt{bool}$
- $\Phi_9 = \Phi_8 \wedge \alpha_3 \leqslant \alpha_3$
- $\Phi_{10} = \Phi_9 \wedge \alpha_6 \leqslant \alpha_6$
- $\Phi_{11} = \Phi_{10} \wedge \alpha_7 \leqslant \alpha_4$
- $\Phi_{12} = \Phi_{11} \wedge \texttt{bool} \leqslant \alpha_4$
- $\Phi_{13} = \Phi_{12} \wedge \alpha_4 \leqslant \alpha_7$
- $\Phi_{14} = \Phi_{13} \wedge \alpha_4 \leqslant \alpha_4$
- $\Phi_{15} = \Phi_{14} \wedge \alpha_7 \leqslant \alpha_7$
- $\Phi_{16} = \Phi_{15} \wedge \alpha_4 \leqslant \alpha \wedge \alpha_7 \leqslant \alpha \wedge \texttt{bool} \leqslant \alpha$
- $\Phi_{17} = \Phi_{15} \wedge \alpha_8 \leqslant \alpha_3 \wedge \alpha_8 \leqslant \alpha_6 \wedge \alpha_8 \leqslant \texttt{bool}$
- $\Phi_{18} = \Phi_{17} \wedge \texttt{bool} \leqslant \alpha_8 \wedge \texttt{bool} \leqslant \alpha_3 \wedge \texttt{bool} \leqslant \alpha_6$

où le sous-arbre $\boxed{\mathtt{T}}$ est :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Phi_8 \vdash \alpha_3 \leq \texttt{bool} \rhd \Phi_8}\,\textsc{STransLeft}}{\Phi_7 \vdash \alpha_3 \leqslant \texttt{bool} \rhd \Phi_8}\,\textsc{SNewConstraint} \quad \dfrac{}{\Phi_9 \vdash \alpha_3 \leq \alpha_3 \rhd \Phi_9}\,\textsc{SSameVar}}{\Phi_8 \vdash \alpha_3 \leqslant \alpha_3 \rhd \Phi_9}\,\textsc{SNewConstraint} \quad \dfrac{}{\Phi_{10} \vdash \alpha_6 \leq \alpha_6 \rhd \Phi_{10}}\,\textsc{SSameVar}}{\Phi_9 \vdash \alpha_6 \leqslant \alpha_6 \rhd \Phi_{10}}\,\textsc{SNewConstraint}}{\Phi_7 \vdash \alpha_3 \leq \alpha_6 \rhd \Phi_{10}}\,\textsc{SDiffVar} \quad \dfrac{\dfrac{\dfrac{}{\Phi_6 \vdash \alpha_6 \leq \alpha_3 \rhd \Phi_6}\,\textsc{SDiffVar}}{\Phi_5 \vdash \alpha_6 \leqslant \alpha_3 \rhd \Phi_6}\,\textsc{SNewConstraint}}{\Phi_6 \vdash \alpha_3 \leqslant \alpha_6 \rhd \Phi_{10}}\,\textsc{SNewConstraint}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Phi_{12} \vdash \texttt{bool} \leq \alpha_4 \rhd \Phi_{12}}\,\textsc{STransRight}}{\Phi_{11} \vdash \texttt{bool} \leqslant \alpha_4 \rhd \Phi_{12}}\,\textsc{SNewConstraint} \quad \dfrac{}{\Phi_{14} \vdash \alpha_4 \leq \alpha_4 \rhd \Phi_{14}}\,\textsc{SSameVar}}{\Phi_{13} \vdash \alpha_4 \leqslant \alpha_4 \rhd \Phi_{14}}\,\textsc{SNewConstraint} \quad \dfrac{}{\Phi_{15} \vdash \alpha_7 \leq \alpha_7 \rhd \Phi_{15}}\,\textsc{SSameVar}}{\Phi_{14} \vdash \alpha_7 \leqslant \alpha_7 \rhd \Phi_{15}}\,\textsc{SNewConstraint}}{\Phi_{11} \vdash \alpha_7 \leq \alpha_4 \rhd \Phi_{12}}\,\textsc{SDiffVar} \quad \dfrac{}{\Phi_{13} \vdash \alpha_4 \leq \alpha_7 \rhd \Phi_{15}}\,\textsc{SDiffVar}}{\Phi_{10} \vdash \alpha_7 \leqslant \alpha_4 \rhd \Phi_{12}}\,\textsc{SNewConstraint} \quad \dfrac{}{\Phi_{12} \vdash \alpha_4 \leqslant \alpha_7 \rhd \Phi_{15}}\,\textsc{SNewConstraint}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Phi_5 \vdash \alpha_6 \to \alpha_7 \leq \alpha_3 \to \alpha_4 \rhd \Phi_{15}}\,\textsc{STypeConstr}}{\Phi_4 \vdash \alpha_6 \to \alpha_7 \leqslant \alpha_3 \to \alpha_4 \rhd \Phi_{15}}\,\textsc{SNewConstraint}}{\Phi_4 \vdash \alpha_6 \to \alpha_7 \leq \alpha_5 \rhd \Phi_{15}}\,\textsc{STransRight}}{\Phi_3 \vdash \alpha_6 \to \alpha_7 \leqslant \alpha_5 \rhd \Phi_{15}}\,\textsc{SNewConstraint}$$

La partie « typage » de l'arbre précédent (dont les noms des noeuds sont de la forme T...) représente la décomposition syntaxique de l'expression à typer (en l'occurrence « `not true` »). Le noeud d'instanciation (INST) de gauche correspond à l'instanciation du schéma de type associé à « `not` » et celui de droite à l'instanciation du schéma de type associé à « `true` ». Ces noeuds d'instanciation provoquent la génération de variables de type fraîches : $\alpha_3, \ldots, \alpha_8$. Les feuilles de cet arbre contiennent les différentes inéquations reliant ces variables de type, à la fois entre elles, et avec des types construits (`bool` et $(\rightarrow)$ en l'occurrence).

La partie saturation de l'arbre (composée de noeuds de type S...) cherche quant à elle à prouver la compatibilité des inéquations générées par la partie typage. Elle ne génère pas de nouvelles variables de type mais tente (avec succès, ici) de saturer l'ensemble de contraintes de sous-typage.

À la fin du typage et de la saturation, l'ensemble de contraintes valide et saturé obtenu est Φ_{13}. Les inéquations importantes qu'il contient sont :

- $\texttt{bool} \leqslant \alpha_8 \leqslant \alpha_3 \leqslant \alpha_6 \leqslant \texttt{bool}$: il s'agit de la chaîne reliant le typage de `true` avec le type de l'argument attendu par la primitive `not`. Ces deux types sont compatibles ($\texttt{bool} \leqslant \texttt{bool}$), ce pourquoi la saturation se termine sans erreur.

- $\texttt{bool} \leqslant \alpha_7 \leqslant \alpha_4 \leqslant \alpha$: cette multi-inéquation résume toutes les contraintes accumulées sur α lors du typage de `(not true :` α`)`.

Le schéma de type que l'algorithme d'inférence donne pour l'expression `(not true)` est $\textsc{gen}(\alpha, \Phi_{13}, \varnothing)$. La définition naïve de GEN nous renvoie le schéma de type :

$$[\ \forall \alpha\ \alpha_3\ \alpha_5\ \alpha_6\ \alpha_7\ \alpha_8\ .\ \alpha \mid \Phi_{13}\ ]$$

qui est tout à fait correct, mais « pollué » de variables de type et d'inéquations inutiles. Un simple nettoyage ne retenant que les variables de type et inéquations liées à α donne le schéma de type :

$$[\ \forall \alpha\ \alpha_4\ \alpha_7\ .\ \alpha \mid \texttt{bool} \leqslant \alpha \wedge \texttt{bool} \leqslant \alpha_4 \wedge \texttt{bool} \leqslant \alpha_7 \wedge \alpha_4 \leqslant \alpha \wedge \alpha_4 \leqslant \alpha_7 \wedge \alpha_7 \leqslant \alpha \wedge \alpha_7 \leqslant \alpha_4\ ]$$

Un nettoyage avancé (tenant compte de la parité des variables de type) retirera alors l'inéquation $\alpha_4 \leqslant \alpha_7$. Un tel nettoyage est possible puisque, à chaque instanciation de ce schéma, la variable de type associée à α après renommage se retrouvera à gauche d'un $(\leqslant)$ (il suffit de se référer à la définition de la règle INST pour s'en convaincre). L'inéquation obtenue après renommage de $\alpha_4 \leqslant \alpha_7$ ne pourra donc jamais faire partie d'une chaîne d'inéquations reliant un τ^l à un τ^r et donc jamais engendrer un clash de typage. Après cette simplification, nous obtenons le schéma de type :

$$[\ \forall \alpha\ \alpha_4\ \alpha_7\ .\ \alpha \mid \texttt{bool} \leqslant \alpha \wedge \texttt{bool} \leqslant \alpha_4 \wedge \texttt{bool} \leqslant \alpha_7 \wedge \alpha_4 \leqslant \alpha \wedge \alpha_7 \leqslant \alpha \wedge \alpha_7 \leqslant \alpha_4\ ]$$

Enfin, comme les variables α_4 et α_7 sont généralisées et n'apparaissent que dans la chaîne reliant `bool` à α, un algorithme de simplification intelligent peut les supprimer sans perdre, ni en généralité, ni en validité. Le schéma de type finalement obtenu est :

$$[\ \forall\alpha\ .\ \alpha \mid \texttt{bool} \leqslant \alpha\ ]$$

Ce schéma de type est en particulier égal (à α-conversion près) à T`(false)`.

Remarque : la taille de l'arbre inféré précédemment peut surprendre vis-à-vis de la simplicité de l'expression analysée (à savoir « `not true` »). Il est néanmoins important de noter que toutes les étapes de l'algorithme d'inférence ont été détaillées et que le « principe » du déroulage de cet arbre est simple et standard : pour chaque noeud, soit aucune règle ne peut s'appliquer, l'arbre est alors non-constructible et le typage s'arrête ; soit une unique règle peut s'appliquer et la construction continue mécaniquement.

Les règles de typage et de saturation qui ont été décrites dans cette section constituent donc un moyen précis d'inférer un schéma de type pour une expression. En plus de ce schéma de type, ces règles nous permettent de générer un « arbre d'inférence » constituant une « preuve de typage » de notre programme. Il reste néanmoins à démontrer que la construction de cet arbre se termine toujours, et que les expressions acceptées par un tel « typeur » s'exécuteront effectivement correctement. C'est le but de la prochaine section.

2.4 Validité du typage

Le mécanisme que nous avons décrit précédemment pour définir les systèmes de types consiste en la construction d'un « arbre d'inférence » à partir d'un ensemble de règles d'inférence. Si ce système de types est implémenté, nous obtenons un algorithme d'inférence permettant d'inférer un schéma de type pour des expressions. Il serait de bon goût que cet algorithme se termine, quel que soit le programme qui lui est donné en entrée. Pour ce faire, nous allons démontrer qu'un arbre d'inférence est toujours de taille finie.

D'autre part, nous avons jusqu'à maintenant défini la sémantique de notre langage complètement indépendamment de son système de types. Rien ne prouve qu'ils sont « compatibles », au sens où si le système de types accepte un programme, celui-ci s'exécutera correctement. Une telle propriété se nomme « validité du système de types » (sous-entendu vis-à-vis d'une sémantique). Nous allons donc formaliser ici ce théorème, puis donner le schéma général de preuve que nous utiliserons pour le prouver sur les différents systèmes de types que nous étudions dans cette thèse, et enfin le prouver pour le système de types de base que nous avons défini dans la section précédente.

2.4.1 Définitions préliminaires

L'énoncé et la preuve des théorèmes de terminaison et de validité nécessite quelques définitions préliminaires sur les ensembles de contraintes et les schémas de type.

Propriétés des ensembles de contraintes

Nous commençons par définir la notion de « validité d'une contrainte de sous-typage » vis-à-vis d'un système de types :

Définition 1 (Validité d'une contrainte « $\tau^l \leqslant \tau^r$ »)**.**

Une contrainte de type $\tau^l \leqslant \tau^r$ est dite « valide » s'il existe une règle de saturation dont la conclusion est de la forme $\Phi \vdash \tau^l \leq \tau^r \rhd \Phi'$, autrement dit s'il est possible d'effectuer au moins un pas de saturation à partir de $\Phi \vdash \tau^l \leq \tau^r \rhd \Phi'$.

Avec le système de types de base que nous avons défini précédemment, toute relation entre une variable de type et un type est valide. Par contre, certaines relations entre types construits ne sont pas valides comme par exemple « `int` $\leqslant$ `string` » ou « `bool` $\leqslant \alpha_1 \times \alpha_2$ » ou encore « $\alpha_1 \to \alpha_2 \leqslant \{$ `A ‖ B` $\}$ ».
Nous étendons cette définition à un ensemble de contraintes :

Définition 2 (Validité d'un ensemble « Φ » de contraintes)**.**

Un ensemble de contraintes Φ est dit « valide » si toutes ses contraintes sont valides.

La simple « validité » d'un ensemble de contraintes Φ n'est pas très intéressante en soi puisque cette notion n'empêche pas Φ de contenir des contraintes qui pourraient impliquer des contraintes invalides. Pour éviter cela, nous allons maintenant définir la notion de « saturation » d'un ensemble de contraintes :

Définition 3 (Saturation d'un ensemble « Φ » de contraintes)**.**

Un ensemble de contraintes Φ est dit « saturé » s'il satisfait toutes les propriétés suivantes :

- $\forall \alpha, \tau^l, \tau^r \;.\; (\tau^l \leqslant \alpha) \in \Phi \wedge (\alpha \leqslant \tau^r) \in \Phi \Rightarrow$
  $\wedge\; (\tau^l \leqslant \tau^r) \in \Phi$
- $\forall \mathtt{t}, \alpha_1, \ldots, \alpha_n, \alpha'_1, \ldots, \alpha'_n \;.\; ((\alpha_1, \ldots, \alpha_n)\ \mathtt{t} \leqslant (\alpha'_1, \ldots, \alpha'_n)\ \mathtt{t}) \in \Phi \Rightarrow$
  $\wedge\; (\alpha_1 \leqslant \alpha'_1) \in \Phi \wedge \ldots \wedge (\alpha_n \leqslant \alpha'_n) \in \Phi$
  $\wedge\; (\alpha'_1 \leqslant \alpha_1) \in \Phi \wedge \ldots \wedge (\alpha'_n \leqslant \alpha_n) \in \Phi$
- $\forall K, \alpha, \alpha' \;.\; (K\,\alpha \leqslant \{\ \ldots\ \|\ K\,\alpha'\ \|\ \ldots\ \}) \in \Phi \Rightarrow$
  $\wedge\; (\alpha \leqslant \alpha') \in \Phi$
- $\forall K, K_1, \ldots, K_n, \alpha, \alpha_d, \alpha_1, \ldots, \alpha_n \;.$
  $(K\,\alpha \leqslant \{\ K_1\ \alpha_1\ \|\ \ldots\ \|\ K_n\ \alpha_n\ \|\ \alpha_d\ \}) \in \Phi \wedge (\forall i \;.\; K \neq K_i) \Rightarrow$
  $\wedge\; (K\,\alpha \leqslant \alpha_d) \in \Phi$
- $\forall t, \alpha_d, \alpha_1, \ldots, \alpha_n \;.\; ((\alpha_1, \ldots, \alpha_n)\ t \leqslant \{\ \ldots\ \|\ \alpha_d\ \}) \in \Phi \Rightarrow$
  $\wedge\; ((\alpha_1, \ldots, \alpha_n)\ t \leqslant \alpha_d) \in \Phi$

Tout comme la validité, la saturation d'un ensemble de contraintes est directement liée au système de types. Ces deux notions évolueront donc au cours des chapitres de cette thèse.

Nous définissons maintenant une relation, notée $(\overset{\alpha}{=})$ représentant l'égalité modulo α-renommage entre deux ensembles de contraintes :

Définition 4 (Égalité de deux ensembles de contraintes modulo α-renommage)**.**

Deux ensembles de contraintes Φ et Φ' vérifient la relation $\Phi \overset{\alpha}{=} \Phi'$ s'il existe une fonction totale de renommage R des variables de type qui soit telle que $R(\Phi') = \Phi$.

Par ailleurs, nous définissons une relation d'ordre notée $(\leqslant)$ entre deux ensembles de contraintes :

Définition 5 (Relation d'ordre sur les ensembles de contraintes)**.**

Deux ensembles de contraintes Φ et Φ' vérifient $\Phi \leqslant \Phi'$ s'il existe une fonction totale de renommage R des variables de type de telle sorte que $R(\Phi) \subset \Phi'$.

Il est important de remarquer que la fonction de renommage R n'est pas supposée injective. Deux variables de type peuvent donc avoir la même image par R.

La relation ($\leqslant$) est bien une relation d'ordre vis-à-vis de la relation d'égalité ($\overset{\alpha}{=}$). En effet, elle est dotée des propriétés suivantes :

- réflexivité : tout ensemble de contraintes Φ vérifie $\Phi \leqslant \Phi$. Il suffit de prendre la fonction de renommage identité pour le démontrer
- transitivité : si $\Phi_1 \leqslant \Phi_2$ et $\Phi_2 \leqslant \Phi_3$ alors $\Phi_1 \leqslant \Phi_3$. La démonstration est évidente, il suffit de choisir la composée des deux fonctions de renommage
- anti-symétrie : si $\Phi_1 \leqslant \Phi_2$ et $\Phi_2 \leqslant \Phi_1$ alors $\Phi_1 \overset{\alpha}{=} \Phi_2$. Un petit raisonnement sur les cardinaux montre que les fonctions de renommage sont ici des bijections, et donc que les ensembles de contraintes Φ_1 et de Φ_2 ont obligatoirement le même cardinal. Les inclusions entre ensembles de contraintes sont donc des égalités.

Nous noterons $\Phi_1 \leqslant_R \Phi_2$ pour spécifier la fonction de renommage utilisée.

Cette relation d'ordre sera importante pour énoncer les « lemmes de réduction du sujet » spécifiant que l'ensemble de contraintes généré par le typage d'une expression « décroît » lorsque l'on évalue l'expression.

Propriétés des schémas de type

Les définitions de validité et de saturation d'un ensemble de contraintes s'étendent naturellement aux schémas de type :

Définition 6 (Validité d'un schéma de type)**.**

Un schéma de type $[\ \forall\alpha_1 \dots \alpha_n\ .\ \alpha_0 \mid \Phi\ ]$ *est « valide » si* Φ *est valide.*

Définition 7 (Saturation d'un schéma de type)**.**

Un schéma de type $[\ \forall\alpha_1 \dots \alpha_n\ .\ \alpha_0 \mid \Phi\ ]$ *est « saturé » si* Φ *est saturé.*

Nous étendons de la même manière la relation d'égalité modulo α-renommage ($\overset{\alpha}{=}$) aux schémas de type :

Définition 8 (Égalité de deux schémas de type modulo α-renommage)**.**

Deux schémas de type $[\ \forall\alpha_1 \dots \alpha_n\ .\ \alpha_0 \mid \Phi\ ]$ *et* $[\ \forall\alpha'_1 \dots \alpha'_{n'}\ .\ \alpha'_0 \mid \Phi'\ ]$ *vérifient la relation* $[\ \forall\alpha_1 \dots \alpha_n\ .\ \alpha_0 \mid \Phi\ ] \overset{\alpha}{=} [\ \forall\alpha'_1 \dots \alpha'_{n'}\ .\ \alpha'_0 \mid \Phi'\ ]$ *s'il existe une fonction totale de renommage R des variables de type qui soit telle que :*

- $R(\alpha_0) = \alpha'_0$
- $R(\{\ \alpha_1, \dots, \alpha_n\ \}) = \{\ \alpha'_1, \dots, \alpha'_{n'}\ \}$
- $\forall\alpha\ .\ \alpha \notin \{\ \alpha_1, \dots, \alpha_n\ \} \Rightarrow R(\alpha) = \alpha$
- $R(\Phi) = \Phi'$

De même, nous étendons la comparaison ($\leqslant$) aux schémas de type :

Définition 9 (Relation d'ordre sur les schémas de type)**.**

Deux schémas de type $[\ \forall\alpha_1 \dots \alpha_n\ .\ \alpha_0 \mid \Phi\ ]$ *et* $[\ \forall\alpha'_1 \dots \alpha'_{n'}\ .\ \alpha'_0 \mid \Phi'\ ]$ *vérifient la relation* $[\ \forall\alpha_1 \dots \alpha_n\ .\ \alpha_0 \mid \Phi\ ] \leqslant [\ \forall\alpha'_1 \dots \alpha'_{n'}\ .\ \alpha'_0 \mid \Phi'\ ]$ *s'il existe une fonction totale de renommage R des variables qui soit telle que :*

- $R(\alpha_0) = \alpha'_0$
- $R(\{\ \alpha_1, \dots, \alpha_n\ \}) \subset \{\ \alpha'_1, \dots, \alpha'_{n'}\ \}$
- $\forall\alpha\ .\ \alpha \notin \{\ \alpha_1, \dots, \alpha_n\ \} \Rightarrow R(\alpha) = \alpha$
- $R(\Phi) \subset \Phi'$

Tout comme pour les comparaisons entre ensembles de contraintes, la fonction de renommage R n'est pas supposée injective.

Cette relation d'ordre entre schémas de type est construite pour représenter l'inverse de l'inclusion des ensembles de valeurs qu'ils dénotent. En effet, imposer $R(\Phi) \subset \Phi'$ autorise Φ' à avoir « plus de contraintes » que Φ. Puisque Φ et Φ' sont des conjonctions de contraintes, avoir « plus de contraintes » permet effectivement de dénoter un ensemble plus petit. Par exemple, ce que l'on exprime habituellement par « le type $(\alpha \to \alpha)$ est un sous-type de $(\texttt{int} \to \texttt{int})$ » s'écrira dans notre langage :

$$[\ \forall\alpha_0\ \alpha\ .\ \alpha_0 \mid \alpha \to \alpha \leqslant \alpha_0\ ] \ \leqslant\ [\ \forall\alpha_0\ \alpha\ .\ \alpha_0 \mid \alpha \to \alpha \leqslant \alpha_0\ \wedge\ \texttt{int} \leqslant \alpha\ \wedge\ \alpha \leqslant \texttt{int}\ ]$$

De plus, le fait que la fonction de renommage R ne soit pas supposée injective autorise des variables de type distinctes de $[\ \forall\alpha'_1 \dots \alpha'_{n'}\ .\ \alpha'_0 \mid \Phi'\ ]$ à avoir leur variables correspondantes égales dans $[\ \forall\alpha_1 \dots \alpha_n\ .\ \alpha_0 \mid \Phi\ ]$. Par exemple, ce que l'on exprime habituellement par « le type $(\alpha_1 \to \alpha_2)$ est un sous-type de $(\alpha \to \alpha)$ » s'écrira dans notre langage :

$$[\ \forall\alpha_0\ \alpha_1\alpha_2\ .\ \alpha_0 \mid \alpha_1 \to \alpha_2 \leqslant \alpha_0\ ] \ \leqslant\ [\ \forall\alpha_0\ \alpha\ .\ \alpha_0 \mid \alpha \to \alpha \leqslant \alpha_0\ ]$$

La fonction R à choisir pour montrer une telle inégalité est définie par :

- $R(\alpha_0) = \alpha_0$
- $R(\alpha_1) = R(\alpha_2) = \alpha_1$ (non-injectivité de R)
- $\forall\alpha' \notin \{\ \alpha_0, \alpha_1, \alpha_2\ \}\ .\ R(\alpha') = \alpha'$

En réalité, cette relation ($\leqslant$) est strictement plus forte que l'inverse de l'inclusion des ensembles : si deux schémas de type σ_1 et σ_2 vérifient $\sigma_1 \leqslant \sigma_2$, alors les ensembles de valeurs qu'ils dénotent V_1 et V_2 vérifient $V_2 \subset V_1$ mais l'inverse n'est pas toujours vrai. Par exemple, les schémas de type :

- $\sigma_1 = [\ \forall\alpha_0\ \alpha\ .\ \alpha_0 \mid \alpha \to \alpha \leqslant \alpha_0\ ]$
- $\sigma_2 = [\ \forall\alpha_0\ \alpha_1\ \alpha_2\ .\ \alpha_0 \mid \alpha_1 \to \alpha_2 \leqslant \alpha_0\ \wedge\ \alpha_1 \leqslant \alpha_2\ \wedge\ \alpha_2 \leqslant \alpha_1\ ]$

dénotent les mêmes ensembles mais ne vérifient ni $\sigma_1 \leqslant \sigma_2$ ni $\sigma_2 \leqslant \sigma_1$.

Propriétés des environnements de typage

Nous étendons ici la relation de comparaison ($\leqslant$) aux environnements de typage :

Définition 10 (Relation d'ordre sur les environnements de typage)**.**

Deux environnements de typage Γ_1 et Γ_2 vérifient la relation $\Gamma_1 \leqslant \Gamma_2$ s'ils ont exactement le même domaine de variables X et si :

$$\forall x \in X \,.\, \Gamma_1[x] \leqslant \Gamma_2[x]$$

Relation entre une expression et un schéma de type

Étant donné un système de types, nous allons maintenant définir une relation notée $(:)$ entre une expression et un schéma de type :

Définition 11 (« $\mathsf{e} \ : \ \sigma$ »)**.**

Une expression e et un schéma de type σ vérifient la relation $\mathsf{e} : \sigma$ si, étant donnée une variable de type fraîche α, les deux propriétés suivantes sont vérifiées :

- $\varnothing, \varnothing \vdash \mathsf{e} : \alpha \rhd \Phi$
- $\textsc{gen}(\alpha, \Phi, \varnothing) \leqslant \sigma$

Nous remarquerons que, pour toute expression e, s'il existe un σ tel que $\mathsf{e} : \sigma$, alors une infinité de σ vérifient $\mathsf{e} : \sigma$. De plus, on dira qu'une expression e est « typable » si et seulement s'il existe au moins un σ tel que $\mathsf{e} : \sigma$.

2.4.2 Terminaison

Nous cherchons ici à démontrer que les algorithmes d'inférence que nous définissons dans cette thèse se terminent. Une telle preuve utilisera très peu de propriétés du système de types et sera donc presque identique pour tous les systèmes étudiés dans cette thèse.

L'algorithme d'inférence consiste à construire l'arbre d'inférence grâce aux règles de typage, d'instanciation et de saturation du système de types. Nous allons simplement prouver qu'un tel arbre est toujours de taille finie.

En premier lieu, il est important de noter que la répartition des noeuds de typage et de saturation n'est pas « quelconque » dans un arbre d'inférence. Ceci est dû à deux propriétés :

- La racine de l'arbre est toujours un noeud de typage.
- Aucune règle de saturation n'a en prémisse la conclusion d'une règle de typage.

Par conséquent, la partie de l'arbre d'inférence constituée de noeuds de typage forme un arbre dont les feuilles sont soit des noeuds d'instanciation, soit les racines de sous-arbres constitués uniquement de noeuds de saturation. Nous appellerons par la suite « sous-arbre de saturation » un sous-arbre maximal composé uniquement d'instances de règles de saturation. Les noeuds d'instanciation ont également en prémisse des conclusion de règles de saturation. Les prémisses des noeuds d'instanciation sont donc également les racines de « sous-arbres de saturation ».

Pour prouver que tout arbre d'inférence est de taille finie, nous allons commencer par prouver que la « partie typage » de l'arbre est de taille finie, puis que le nombre de noeuds d'instanciation est fini, et enfin que tous les « sous-arbres de saturation » sont de taille finie.

Étude de la « partie typage »

Nous démontrons ici que le nombre de noeuds de typage d'un arbre d'inférence est égal au nombre de noeuds de l'expression que l'on cherche à typer.

Il est facile de vérifier que pour chaque règle de typage de nos systèmes de types, chacune de ses prémisses est :

- soit la conclusion d'une règle de saturation
- soit la conclusion d'une règle de typage dont l'expression est une sous-expression de celle présente dans la conclusion de la règle de typage
- soit la conclusion de la règle Inst.

La « partie typage » de l'arbre d'inférence est par conséquent isomorphe à l'arbre de l'expression analysée. Il possède en particulier autant de noeuds. L'expression que l'on cherche à typer étant finie, la partie typage de l'arbre d'inférence est donc finie.

Étude des noeuds d'instanciation

Les seules règles d'inférence ayant en prémisse la conclusion d'une règle d'instanciation sont les règles de typage TConst, TApplyPrim1, TApplyPrim2 et TVar. Le nombre de noeuds d'instanciation dans un arbre de typage est alors inférieur ou égal au nombre de noeuds de typage. Comme prouvé précédemment, les noeuds de typage sont en nombre fini dans l'arbre, et chacun engendre au plus un noeud d'instanciation. Le nombre d'occurrences de noeuds d'instanciation est donc fini.

Étude des sous-arbres de saturation

Nous avons déjà prouvé que les sous-arbres de saturation sont en nombre fini puisqu'ils sont engendrés par des noeuds de typage et d'instanciation qui sont en nombre fini. Nous démontrons ici que leur taille est finie.

Nous remarquons qu'aucune règle de saturation ne génère de nouvelles variables de type. Le nombre de variables de type apparaissant dans un sous-arbre de saturation est donc fini. De

plus, le nombre de constructeurs de types est également fini ainsi que leur arité. En effet, les constructeurs de types sont pour l'instant uniquement prédéfinis (`int`, `bool`, $(\rightarrow)$, $(\times)$, etc.), et ils seront enrichis dans le chapitre 5 sur les **GADT** avec un nombre fini de constructeurs de types déclarés dans le programme par l'utilisateur. Comme les définitions de τ^l et de τ^r ne sont pas récursives, il existe un ensemble fini de τ^l et de τ^r différents dans le sous-arbre de saturation, et donc un ensemble fini de contraintes de type de la forme $\tau^l \leqslant \tau^r$.

Chaque occurrence de la règle SNewConstraint ajoute une contrainte de type dans Φ qui n'y est pas encore. Aucune règle ne supprime de contrainte de Φ. Le nombre de contraintes étant borné dans le sous-arbre de saturation, la taille de Φ est bornée, et le nombre d'occurrences de la règle SNewConstraint est donc fini.

La règle SNewConstraint est la seule règle de saturation à avoir une prémisse de la forme $\Phi \vdash \tau^l \leq \tau^r \rhd \Phi'$. Comme le nombre d'occurrences de SNewConstraint est fini, le nombre d'occurrences de conclusions de la forme $\Phi \vdash \tau^l \leq \tau^r \rhd \Phi'$ est fini. Par conséquent, toutes les règles de saturation ayant une conclusion de la forme $\Phi \vdash \tau^l \leq \tau^r \rhd \Phi'$ (à savoir STypeConstr, SVariantMatch, SVariantDefault, SConstrDefault, SSameVar, STransRight, STransLeft et STransLeftRight) ont un nombre fini d'occurrences dans le sous-arbre de saturation.

Enfin, la seule règle de saturation dont on n'a pas encore prouvé que son nombre d'occurrences dans le sous-arbre de saturation était fini est SAlreadyProved. Il s'agit d'un axiome qui ne peut donc apparaître qu'au niveau d'une feuille. Nous avons donc montré que le sous-arbre de saturation a un intérieur fini, on en conclut donc que le sous-arbre de saturation lui-même est fini.

2.4.3 Théorèmes de validité

Nous souhaitons ici énoncer formellement un « théorème de validité » signifiant que si un programme est « accepté » par le typeur, alors son exécution se passera correctement :

Théorème 1 (Validité faible)**.**

Pour toute expression `e` *et tout schéma de type* σ*, si* `e` *et* σ *vérifient* `e` $: \sigma$ *alors* `eval(e)` $\neq$ `ERROR`*.*

En réalité, nous allons énoncer un second théorème, plus fort que le premier, permettant en plus, lorsque qu'une expression `e` est typable et que son exécution se termine, de relier le résultat de l'exécution au schéma de type inféré pour `e` :

Théorème 2 (Validité forte)**.**

Pour toute expression `e` *et tout schéma de type* σ*, si* `e` *et* σ *vérifient* `e` $: \sigma$ *alors l'une des propriétés suivantes est vérifiée :*

- *L'évaluation de* `e` *boucle indéfiniment.*
- *L'expression* `e` *s'évalue en la valeur* `v` *(c'est-à-dire* `e` $\longmapsto\!\!\!\!\!\rightarrow$ `v`*) et* `v` $: \sigma$*.*

Il n'est pas nécessaire d'imposer, dans ces théorèmes, que σ soit valide et saturé car la définition de $\mathsf{e} : \sigma$ suffit à imposer qu'il existe un σ' (potentiellement différent de σ) valide et saturé tel que $\mathsf{e} : \sigma'$. La propriété 1 que nous allons voir nous montrera qu'il suffit de choisir $\sigma' = \textsc{gen}(\alpha, \Phi, \varnothing)$.

2.4.4 Schéma des preuves

Les preuves des théorèmes de validité que nous allons donner sont inspirées de celles présentées par Wright et Felleisen [WF92]. Nous en donnons ici le schéma général. Une telle démonstration s'obtient en montrant les lemmes suivants.

Préservation des propriétés de Φ

Nous commençons par énoncer deux lemmes mettant en évidence la préservation des propriétés de « validité » et de « saturation » de l'ensemble de contraintes Φ lors du typage d'une expression.

Lemme 1 (Préservation de la validité de Φ)**.**

Soient :

- Γ *un environnement de typage ne contenant que des schémas de type valides*
- Φ *un ensemble de contraintes valides*
- e *une expression*
- α *une variable de type*
- Φ' *l'ensemble de contraintes obtenu par le déroulage réussi de l'arbre d'inférence en partant de la racine* $(\Phi, \Gamma \vdash \mathsf{e} : \alpha \rhd \Phi')$*.*

Alors Φ' *est valide.*

La démonstration de ce lemme découle directement de la définition de « validité d'une contrainte ».

Le lemme de préservation de la saturation de Φ est très similaire :

Lemme 2 (Préservation de la saturation de Φ)**.**

Soient :

- Γ *un environnement de typage ne contenant que des schémas de type saturés*
- Φ *un ensemble de contraintes saturé*
- e *une expression*
- α *une variable de type*
- Φ' *l'ensemble de contraintes obtenu par le déroulage réussi de l'arbre d'inférence en partant de la racine* $(\Phi, \Gamma \vdash \mathsf{e} : \alpha \rhd \Phi')$

Alors Φ' *est saturé.*

De ces deux lemmes se déduit une propriété importante pour le système de types :

Propriété 1 (Cohérence du système de types)**.**

Le typage d'une expression, lorsqu'il réussit, génère un schéma de type valide et saturé. Autrement dit, si Φ est obtenu par construction d'un arbre d'inférence ayant pour racine :

$$\varnothing, \varnothing \vdash \mathsf{e} : \alpha \rhd \Phi$$

alors $\textsc{gen}(\alpha, \Phi, \varnothing)$ *est valide et saturé.*

Les deux lemmes suivants mettent en évidence comment évolue l'ensemble de contraintes généré par le typeur lorsque l'on réduit l'ensemble de contraintes fourni :

Lemme 3 (Monotonie de la saturation)**.**

Soient :

- $(\tau^l \leqslant \tau^r)$ *une contrainte de types*
- Φ_1 *et* Φ_2 *deux ensembles de contraintes vérifiant* $\Phi_2 \leqslant \Phi_1$
- Φ'_1 *l'ensemble de contraintes obtenu par saturation de* $(\tau^l \leqslant \tau^r)$ *dans* Φ_1 *en dérivant l'arbre d'inférence au dessus de* $\Phi_1 \vdash \tau^l \leqslant \tau^r \rhd \Phi'_1$.

Alors le déroulage de l'arbre d'inférence au dessus de :

$$\Phi_2 \vdash \tau^l \leqslant \tau^r \rhd \Phi'_2$$

réussit et génère un ensemble de contraintes Φ'_2 *vérifiant* $\Phi'_2 \leqslant \Phi'_1$.

Lemme 4 (Monotonie du typage)**.**

Soient :

- e *une expression*
- α *une variable de type*
- R *une fonction de renommage des variables de type*
- Φ_1 *et* Φ_2 *deux ensembles de contraintes vérifiant* $\Phi_2 \leqslant_R \Phi_1$
- Γ_1 *et* Γ_2 *deux environnements de typage vérifiant* $\Gamma_2 \leqslant_R \Gamma_1$
- Φ'_1 *l'ensemble de contraintes obtenu par typage de* $\mathsf{e} : \alpha$ *dans* Φ_1, Γ_1 *en dérivant l'arbre d'inférence au dessus de* $\Phi_1, \Gamma_1 \vdash \mathsf{e} : \alpha \rhd \Phi'_1$.

Alors le déroulage de l'arbre d'inférence au dessus de :

$$\Phi_2, \Gamma_2 \vdash \mathsf{e} : \alpha \rhd \Phi'_2$$

réussit et génère un ensemble de contraintes Φ'_2 *vérifiant* $\Phi'_2 \leqslant \Phi'_1$.

Nous remarquerons que pour que ce lemme soit correct, la fonction de renommage liant Φ_1 et Φ_2 doit être la même que celle liant Γ_1 et Γ_2.

Lemmes de « réduction du sujet »

Les trois lemmes suivants mettent en évidence la préservation de la typabilité lors de l'évaluation d'une expression par les différentes relations $(\longrightarrow)$, $(\longmapsto)$ et $(\longmapsto\!\!\!\!\!\rightarrow)$.

Lemme 5 (Réduction du sujet par $(\longrightarrow)$)**.**

Soient :

- e_1 *et* e_2 *deux expressions vérifiant* $e_1 \longrightarrow e_2$
- α *une variable de type*
- Φ *un ensemble de contraintes valide et saturé*
- Φ'_1 *l'ensemble de contraintes obtenu par typage de* $e_1 : \alpha$ *dans* $(\Phi, \varnothing)$ *en dérivant l'arbre d'inférence au dessus de* $\Phi, \varnothing \vdash e_1 : \alpha \rhd \Phi'_1$.

Alors le typage de $e_2 : \alpha$ *dans* $(\Phi, \varnothing)$ *réussit et l'ensemble de contraintes* Φ'_2 *obtenu en dérivant l'arbre d'inférence au dessus de* $\Phi, \varnothing \vdash e_2 : \alpha \rhd \Phi'_2$ *vérifie* $\Phi'_2 \leqslant \Phi'_1$.

La démonstration de ce lemme consiste à analyser les différents cas de la définition de $(\longrightarrow)$, et pour chaque réduction de la forme $e_1 \longrightarrow e_2$, construire l'arbre d'inférence de $\Phi, \varnothing \vdash e_2 : \alpha \rhd \Phi'_2$ par transformation de l'arbre d'inférence de $\Phi, \varnothing \vdash e_1 : \alpha \rhd \Phi'_1$ et montrer $\Phi'_2 \leqslant \Phi'_1$.

On étend ensuite ce lemme de réduction du sujet à $(\longmapsto)$ ainsi :

Lemme 6 (Réduction du sujet par $(\longmapsto)$)**.**

Soient :

- e_1 *et* e_2 *deux expressions vérifiant* $e_1 \longmapsto e_2$
- α *une variable de type*
- Φ *un ensemble de contraintes valide et saturé*
- Φ'_1 *l'ensemble de contraintes obtenu par typage de* $e_1 : \alpha$ *dans* $(\Phi, \varnothing)$ *en dérivant l'arbre d'inférence au dessus de* $\Phi, \varnothing \vdash e_1 : \alpha \rhd \Phi'_1$.

Alors le typage de $e_2 : \alpha$ *dans* $(\Phi, \varnothing)$ *réussit et l'ensemble de contraintes* Φ'_2 *obtenu en dérivant l'arbre d'inférence au dessus de* $\Phi, \varnothing \vdash e_2 : \alpha \rhd \Phi'_2$ *vérifie* $\Phi'_2 \leqslant \Phi'_1$.

La démonstration de ce lemme se fait par induction sur la structure du contexte d'évaluation et une analyse de cas sur la définition de la grammaire E.

On étend enfin ce lemme à $(\longmapsto\!\!\!\!\!\rightarrow)$ ainsi :

Lemme 7 (Réduction du sujet par $(\longmapsto\!\!\!\!\!\rightarrow)$)**.**

Soient :

- e_1 *et* e_2 *deux expressions vérifiant* $e_1 \longmapsto\!\!\!\!\!\rightarrow e_2$.
- α *une variable de type*
- Φ *un ensemble de contraintes valide et saturé*
- Φ'_1 *l'ensemble de contraintes obtenu par typage de* $e_1 : \alpha$ *dans* $(\Phi, \varnothing)$ *en dérivant l'arbre d'inférence au dessus de* $\Phi, \varnothing \vdash e_1 : \alpha \rhd \Phi'_1$.

Alors le typage de $e_2 : \alpha$ *dans* $(\Phi, \varnothing)$ *réussit et l'ensemble de contraintes* Φ'_2 *obtenu en dérivant l'arbre d'inférence au dessus de* $\Phi, \varnothing \vdash e_2 : \alpha \rhd \Phi'_2$ *vérifie* $\Phi'_2 \leqslant \Phi'_1$.

La démonstration est presque immédiate par simple analyse de cas sur la définition de $(\longmapsto\!\!\!\!\!\rightarrow)$.

Propriétés des expressions bloquées

Indépendamment des autres lemmes, on démontre le lemme suivant :

Lemme 8 (Les expressions bloquées ne sont pas typables)**.**

Soit e_b *une expression bloquée. Il n'existe aucun schéma de type* σ *tel que* $e_b : \sigma$.

Ce lemme se démontre en analysant les différentes catégories d'expressions bloquées et en démontrant que pour chacune d'entre elles, il est impossible de construire un arbre d'inférence complet.

Encore indépendamment, on montre le lemme suivant classifiant l'évaluation d'une expression en trois catégories exclusives :

Lemme 9 (Évaluation uniforme)**.**

Toute expression e *vérifie une et une seule des propriétés suivantes :*

- *L'évaluation de* e *boucle indéfiniment*
- *Il existe une valeur* v *telle que* $e \longmapsto\!\!\!\!\!\rightarrow v$
- *Il existe une expression bloquée* e_b *telle que* $e \longmapsto\!\!\!\!\!\rightarrow e_b$.

Ce lemme découle directement de la définition d'une « expression bloquée ».

Preuves des théorèmes de validité

La preuve du théorème de validité forte s'obtient alors en démontrant par l'absurde que s'il existe σ tel que $\mathsf{e} : \sigma$, alors le dernier cas du lemme d'évaluation uniforme est impossible.

Le théorème de validité faible est un simple corollaire du théorème de validité forte.

2.4.5 Preuve de validité du système de base

Nous démontrons ici les différents lemmes et théorèmes énoncés dans le chapitre précédent pour le système de types de base.

Lemme 1 (Préservation de la validité de Φ)**.**

Soient :

- Γ *un environnement de typage ne contenant que des schémas de type valides*
- Φ *un ensemble de contraintes valides*
- e *une expression*
- α *une variable de type*
- Φ' *l'ensemble de contraintes obtenu par le déroulage réussi de l'arbre d'inférence en partant de la racine* $(\Phi, \Gamma \vdash \mathsf{e} : \alpha \rhd \Phi')$*.*

Alors Φ' *est valide.*

Démonstration (Lemme 1) :

La seule règle d'inférence ajoutant une contrainte de la forme $(\tau^l \leqslant \tau^r)$ dans Φ est SNewConstraint, et cette règle possède une prémisse de la forme :

$$\Phi \wedge \tau^l \leqslant \tau^r \vdash \tau^l \leq \tau^r \rhd \Phi'$$

La construction de l'arbre d'inférence a réussi, on en déduit qu'il existe une règle ayant pour conclusion $(\Phi \wedge \tau^l \leqslant \tau^r \vdash \tau^l \leq \tau^r \rhd \Phi')$. L'existence d'une telle règle de saturation correspond bien à la définition de la validité de $(\tau^l \leqslant \tau^r)$.

Lemme 2 (Préservation de la saturation de Φ).

Soient :

- Γ *un environnement de typage ne contenant que des schémas de type saturés*
- Φ *un ensemble de contraintes saturé*
- e *une expression*
- α *une variable de type*
- Φ' *l'ensemble de contraintes obtenu par le déroulage réussi de l'arbre d'inférence en partant de la racine* $(\Phi, \Gamma \vdash \mathsf{e} : \alpha \rhd \Phi')$

Alors Φ' *est saturé.*

Démonstration (Lemme 2) :

Pour montrer que Φ' est saturé, nous montrons indépendamment les différents points de la définition de la « saturation d'un ensemble de contraintes » :

- Soient α, τ^l, τ^r tels que $(\tau^l \leqslant \alpha) \in \Phi'$ et $(\alpha \leqslant \tau^r) \in \Phi'$.
  Montrons $(\tau^l \leqslant \tau^r) \in \Phi'$. Les contraintes $(\tau^l \leqslant \alpha)$ et $(\alpha \leqslant \tau^r)$ ont obligatoirement été ajoutées dans l'ensemble de contraintes par des instances de la règle SNewConstraint l'une après l'autre. On distingue trois cas :
  - Soit les contraintes $(\tau^l \leqslant \alpha)$ et $(\alpha \leqslant \tau^r)$ sont déjà présentes dans Φ. Comme Φ est saturé, $(\tau^l \leqslant \tau^r) \in \Phi$. Aucune règle ne supprime de contrainte, nous avons donc $\Phi \subset \Phi'$ et donc $(\tau^l \leqslant \tau^r) \in \Phi'$.
  - Soit la contrainte $(\tau^l \leqslant \alpha)$ a été ajoutée alors que la contrainte $(\tau^l \leqslant \alpha)$ était déjà présente dans l'ensemble de contraintes. L'application de la règle SNewConstraint sur $(\alpha \leqslant \tau^r)$ a donc généré un noeud de la forme $\Phi_0 \vdash \alpha \leq \tau^r \rhd \Phi_1$ avec $(\tau^l \leqslant \alpha) \in \Phi_0$. La règle appliquée sur $\Phi_0 \vdash \alpha \leq \tau^r \rhd \Phi_1$ est STransLeft. Puisque $(\tau^l \leqslant \alpha) \in \Phi_0$, l'application de STransLeft possède une prémisse de la forme $\Phi'_0 \rhd \tau^l \leqslant \tau^r \vdash \Phi'_1$. Dans cette situation, soit $(\tau^l \leqslant \tau^r)$ était déjà présente dans l'ensemble de contraintes Φ'_0 et la règle SAlreadyProved a été appliquée ; soit $(\tau^l \leqslant \tau^r)$ n'y était pas encore et une instance de la règle SNewConstraint l'y a ajouté. Dans tous les cas, la contrainte $(\tau^l \leqslant \tau^r)$ est bien présente dans l'ensemble de contraintes généré Φ'_1. Comme aucune règle ne supprime de contrainte de l'ensemble de contraintes, Φ'_1 est bien inclus dans Φ'. On en conclut $(\tau^l \leqslant \tau^r) \in \Phi'$.
  - Soit la contrainte $(\tau^l \leqslant \alpha)$ a été ajoutée alors que $(\alpha \leqslant \tau^r)$ était déjà présente dans l'ensemble de contraintes. Un raisonnement similaire basé sur l'utilisation de STransRight à la place de STransLeft prouve également $(\tau^l \leqslant \tau^r) \in \Phi'$.
- Soient $t, \alpha_1, \ldots, \alpha_n, \alpha'_1, \ldots, \alpha'_n$ tels que $((\alpha_1, \ldots, \alpha_n)\, t \leqslant (\alpha'_1, \ldots, \alpha'_n)\, t) \in \Phi'$.
  Montrons que $\forall i \in [\,1;\, n\,]\,.\, (\alpha_i \leqslant \alpha'_i) \in \Phi' \wedge (\alpha'_i \leqslant \alpha_i) \in \Phi'$.
  La seule règle qui a pu ajouter la contrainte $((\alpha_1, \ldots, \alpha_n)\ t \leqslant (\alpha'_1, \ldots, \alpha'_n)\ t)$ dans l'ensemble de contraintes est SNewConstraint et l'instance de cette règle a en prémisse un noeud de la forme $\Phi_0 \vdash ((\alpha_1, \ldots, \alpha_n)\ t \leq (\alpha'_1, \ldots, \alpha'_n)\ t) \rhd \Phi_1$. La seule règle ayant pu

s'appliquer sur un tel noeud est STypeConstr qui a imposé la présence de toutes les contraintes $(\alpha_i \leqslant \alpha'_i)$ et $(\alpha'_i \leqslant \alpha_i)$ pour $i \in [\,1;\, n\,]$ dans l'ensemble contraintes résultat. □

- De la même manière, la propriété « $\forall \mathtt{K}, \alpha, \alpha' \,.\, (\mathtt{K}\ \alpha \leqslant \{\ \ldots\ \|\ \mathtt{K}\ \alpha'\ \|\ \ldots\ \}) \in \Phi \Rightarrow (\alpha \leqslant \alpha') \in \Phi$ » découle directement de l'usage de la règle SVariantMatch lors de l'ajout de la contrainte « $\mathtt{K}\ \alpha \leqslant \{\ \ldots\ \|\ \mathtt{K}\ \alpha'\ \|\ \ldots\ \}$ » dans l'ensemble de contraintes.
- La propriété $\forall \mathtt{K}, \mathtt{K}_1, \ldots, \mathtt{K}_n, \alpha, \alpha_d, \alpha_1, \ldots, \alpha_n \,.\, (\mathtt{K}\ \alpha \leqslant \{\, \mathtt{K}_1\ \alpha_1\ \|\ \ldots\ \|\ \mathtt{K}_n\ \alpha_n\ \|\ \alpha_d\,\}) \in \Phi \Rightarrow (\forall i \,.\, \mathtt{K} \neq \mathtt{K}_i) \Rightarrow (\mathtt{K}\ \alpha \leqslant \alpha_d) \in \Phi$ découle de l'usage de la règle SVariantDefault lors de l'ajout de la contrainte « $\mathtt{K}\ \alpha \leqslant \{\, \mathtt{K}_1\ \alpha_1\ \|\ \ldots\ \|\ \mathtt{K}_n\ \alpha_n\ \|\ \alpha_d\,\}$ » dans l'ensemble de contraintes.
- La propriété $\forall t, \alpha_d, \alpha_1, \ldots, \alpha_n \,.\, ((\alpha_1, \ldots, \alpha_n)\ t \leqslant \{\ \ldots\ \|\ \alpha_d\ \}) \in \Phi \Rightarrow ((\alpha_1, \ldots, \alpha_n)\ t \leqslant \alpha_d) \in \Phi$ découle quant à elle de l'usage de la règle SConstrDefault lors de l'ajout de la contrainte « $(\alpha_1, \ldots, \alpha_n)\ t \leqslant \{\ \ldots\ \|\ \alpha_d\ \}$ » dans l'ensemble de contraintes.

En conclusion, l'ensemble de contraintes Φ' est bien saturé selon tous les critères de la définition de la saturation.

Propriété 1 (Cohérence du système de types)**.**

Le typage d'une expression, lorsqu'il réussit, génère un schéma de type valide et saturé. Autrement dit, si Φ est obtenu par construction d'un arbre d'inférence ayant pour racine :

$$\varnothing, \varnothing \vdash \mathsf{e} : \alpha \rhd \Phi$$

alors $\textsc{gen}(\alpha, \Phi, \varnothing)$ est valide et saturé.

Démonstration (Propriété 1) :

Puisqu'un ensemble de contraintes vide est valide et saturé, cette propriété est une conséquence directe des lemmes 1 et 2.

Lemme 3 (Monotonie de la saturation)**.**

Soient :

- $(\tau^l \leqslant \tau^r)$ *une contrainte de types*
- Φ_1 *et* Φ_2 *deux ensembles de contraintes vérifiant* $\Phi_2 \leqslant \Phi_1$
- Φ'_1 *l'ensemble de contraintes obtenu par saturation de* $(\tau^l \leqslant \tau^r)$ *dans* Φ_1 *en dérivant l'arbre d'inférence au dessus de* $\Phi_1 \vdash \tau^l \leqslant \tau^r \rhd \Phi'_1$.

Alors le déroulage de l'arbre d'inférence au dessus de :

$$\Phi_2 \vdash \tau^l \leqslant \tau^r \rhd \Phi'_2$$

réussit et génère un ensemble de contraintes Φ'_2 *vérifiant* $\Phi'_2 \leqslant \Phi'_1$.

Démonstration (Lemme 3) :

Comme $\Phi_2 \leqslant \Phi_1$, il existe une fonction de renommage R des variables de type telle que $R(\Phi_2) \subset \Phi_1$. Nous construisons alors l'arbre d'inférence de $\Phi_2 \vdash \tau^l \leqslant \tau^r \rhd \Phi'_2$ à partir de l'arbre d'inférence de $\Phi_1 \vdash \tau^l \leqslant \tau^r \rhd \Phi'_1$ en supprimant certains sous-arbres concernant les contraintes de Φ_1 dont leurs correspondantes via R ne sont pas présentes dans Φ_2, et en répliquant certains sous-arbres de Φ_1 concernant les variables de type qui sont fusionnées par la fonction de renommage (non-obligatoirement-injective) R. Le Φ'_2 obtenu après cette transformation vérifie alors $R(\Phi'_2) \subset \Phi'_1$, ce qui montre bien $\Phi'_2 \leqslant \Phi'_1$.

Lemme 4 (Monotonie du typage)**.**

Soient :

- e *une expression*
- α *une variable de type*
- R *une fonction de renommage des variables de type*
- Φ_1 *et* Φ_2 *deux ensembles de contraintes vérifiant* $\Phi_2 \leqslant_R \Phi_1$
- Γ_1 *et* Γ_2 *deux environnements de typage vérifiant* $\Gamma_2 \leqslant_R \Gamma_1$
- Φ'_1 *l'ensemble de contraintes obtenu par typage de* $\mathsf{e} : \alpha$ *dans* Φ_1, Γ_1 *en dérivant l'arbre d'inférence au dessus de* $\Phi_1, \Gamma_1 \vdash \mathsf{e} : \alpha \rhd \Phi'_1$.

Alors le déroulage de l'arbre d'inférence au dessus de :

$$\Phi_2, \Gamma_2 \vdash \mathsf{e} : \alpha \rhd \Phi'_2$$

réussit et génère un ensemble de contraintes Φ'_2 *vérifiant* $\Phi'_2 \leqslant \Phi'_1$.

Démonstration (Lemme 4) :

Nous construisons alors l'arbre d'inférence de $\Phi_2, \Gamma_2 \vdash \mathsf{e} : \alpha \rhd \Phi'_2$ à partir de l'arbre d'inférence de $\Phi_1, \Gamma_1 \vdash \mathsf{e} : \alpha \rhd \Phi'_2$. La partie de cet arbre composée de noeuds de typage reste inchangée car elle est structurellement isomorphe à la structure de l'expression e, qui n'a pas changée. De même, les noeuds d'instanciation restent aux mêmes endroits dans l'arbre puisque leur position est imposée par la partie typage de l'arbre. Comme $\Gamma_2 \leqslant_R \Gamma_1$, les schémas de types instanciés sont plus petits ce qui ne contribue qu'à réduire l'ensemble de contraintes généré. Seuls les sous-arbres de saturation changent de structure. Pour chacun d'entre eux, on applique le lemme 3 montrant qu'on obtient un ensemble de contraintes plus petit (c'est-à-dire ayant moins de contraintes) que l'ensemble de contraintes correspondant dans l'arbre initial. Au final, l'ensemble de contraintes Φ'_2 généré est effectivement plus petit que l'ensemble Φ'_1 initial.

Lemme 5 (Réduction du sujet par $(\longrightarrow)$)**.**

Soient :

- e_1 *et* e_2 *deux expressions vérifiant* $\mathsf{e}_1 \longrightarrow \mathsf{e}_2$
- α *une variable de type*
- Φ *un ensemble de contraintes valide et saturé*
- Φ'_1 *l'ensemble de contraintes obtenu par typage de* $\mathsf{e}_1 : \alpha$ *dans* $(\Phi, \varnothing)$ *en dérivant l'arbre d'inférence au dessus de* $\Phi, \varnothing \vdash \mathsf{e}_1 : \alpha \rhd \Phi'_1$.

Alors le typage de $\mathsf{e}_2 : \alpha$ *dans* $(\Phi, \varnothing)$ *réussit et l'ensemble de contraintes* Φ'_2 *obtenu en dérivant l'arbre d'inférence au dessus de* $\Phi, \varnothing \vdash \mathsf{e}_2 : \alpha \rhd \Phi'_2$ *vérifie* $\Phi'_2 \leqslant \Phi'_1$.

Démonstration (Lemme 5) :

Nous analysons les différents cas de la définition de $(\longrightarrow)$:

- Cas « $\mathsf{p}^1\ \mathsf{v} \longrightarrow \delta_1\ (\mathsf{p}^1, \mathsf{v})$ » :
  La validité du typage repose ici sur la validité de la fonction T vis-à-vis de la fonction d'évaluation des primitives unaires δ_1. En effet, l'arbre d'inférence de $\Phi, \varnothing \vdash \mathsf{p}^1\ \mathsf{v} : \alpha \rhd \Phi'_1$ est obligatoirement de la forme :

$$\textsc{TApplyPrim1}\ \dfrac{\textsc{Inst}\ \dfrac{\bigtriangledown}{\Phi \vdash T(\mathsf{p}^1) \leqslant \alpha_1 \to \alpha_2 \rhd \Phi'} \quad \dfrac{\bigtriangledown}{\Phi' \vdash \alpha_2 \leqslant \alpha \rhd \Phi''} \quad \dfrac{\bigtriangledown}{\Phi'', \varnothing \vdash \mathsf{v} : \alpha_1 \rhd \Phi'_1}}{\Phi, \varnothing \vdash \mathsf{p}^1\ \mathsf{v} : \alpha \rhd \Phi'_1}$$

  Le schéma de type σ retourné par $T(\mathsf{p}^1)$ est instancié et comparé à $\alpha_1 \to \alpha$. La saturation

des contraintes doit alors imposer suffisamment de contraintes sur α_1 pour que si le typage de $\mathtt{v} : \alpha$ réussit, assurer que $\delta_1(\mathtt{p}^1, \mathtt{v})$ soit défini ; et suffisamment de contraintes sur α_2 (lié à α via la contrainte $(\alpha_2 \leqslant \alpha)$) pour que le typage de $\delta_1(\mathtt{p}^1, \mathtt{v}) : \alpha$ réussisse et génère un ensemble de contraintes Φ'_2 plus petit que Φ'_1.

- Cas « $\mathtt{p}^2\ \mathtt{v}_1\ \mathtt{v}_2 \longrightarrow \delta_2\,(\mathtt{p}^2, \mathtt{v}_1, \mathtt{v}_2)$ » :
  L'arbre d'inférence de $\Phi, \varnothing \vdash \mathtt{p}^2\ \mathtt{v}_1\ \mathtt{v}_2\ \mathtt{:}\ \alpha \rhd \Phi'_1$ est obligatoirement de la forme :

$$\textsc{TApplyPrim2}\ \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Phi \vdash \alpha_0 \leqslant \alpha_2 \to \alpha_3 \rhd \Phi_1}\quad \dfrac{}{\Phi_2 \vdash T(\mathtt{p}^2) \leqslant \alpha_1 \to \alpha_0 \rhd \Phi_3}\ \textsc{Inst}\quad \dfrac{}{\Phi_1 \vdash \alpha_3 \leqslant \alpha \rhd \Phi_2}}{\Phi_3, \varnothing \vdash \mathtt{v}_1 : \alpha_1 \rhd \Phi_4}\quad \dfrac{}{\Phi_4, \varnothing \vdash \mathtt{v}_2 : \alpha_2 \rhd \Phi'_1}}{\Phi, \varnothing \vdash \mathtt{p}^2\ \mathtt{v}_1\ \mathtt{v}_2\ \mathtt{:}\ \alpha \rhd \Phi'_1}$$

  De manière similaire au cas de l'application d'un opérateur unaire, le schéma de type généré par $T(\mathtt{p}^2)$ doit imposer suffisamment de contraintes sur α_1 et α_2 pour que si le typage de $\mathtt{v}_1 : \alpha_1$ et $\mathtt{v}_2 : \alpha_2$ réussit, alors $\delta_2(\mathtt{p}^2, \mathtt{v}_1, \mathtt{v}_2)$ soit défini ; et de contraintes sur α de telle sorte que le typage de $\delta_2(\mathtt{p}^2, \mathtt{v}_1, \mathtt{v}_2) : \alpha$ réussisse et génère un ensemble de contraintes Φ'_2 plus petit que Φ'_1.

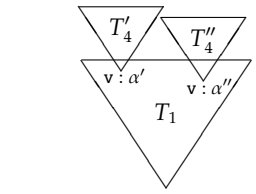
- Cas « $(\lambda\ \mathtt{x}\ .\ \mathtt{e})\ \mathtt{v} \longrightarrow \mathtt{e}[\mathtt{x} \mapsto \mathtt{v}]$ » :
  L'arbre d'inférence de $\Phi, \varnothing \vdash (\lambda\ \mathtt{x}\ .\ \mathtt{e})\ \mathtt{v}\ \mathtt{:}\ \alpha \rhd \Phi'_1$ est obligatoirement de la forme :

$$\textsc{TApp}\ \dfrac{\textsc{STransLeft}\ \dfrac{\textsc{SNewConstraint}\ \dfrac{\Phi_1 \vdash \alpha_1 \leq \alpha_2 \to \alpha_3 \rhd \Phi_1}{\Phi \vdash \alpha_1 \leqslant \alpha_2 \to \alpha_3 \rhd \Phi_1}\quad T_0\ \dfrac{}{\Phi_1 \vdash \alpha_3 \leqslant \alpha \rhd \Phi_2}\quad T_4\ \dfrac{}{\Phi_7, \varnothing \vdash \mathtt{v} : \alpha_2 \rhd \Phi'_1}}{\Phi, \varnothing \vdash (\lambda\ \mathtt{x}\ .\ \mathtt{e})\ \mathtt{v}\ \mathtt{:}\ \alpha \rhd \Phi'_1}$$

$$\textsc{TLambda}\ \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Phi_2, \{\,(\mathtt{x}, \alpha'_1)\,\} \vdash \mathtt{e} : \alpha'_2 \rhd \Phi_3}\quad \dfrac{\dots\quad \dfrac{}{\Phi_4 \vdash \alpha_2 \leqslant \alpha'_1 \rhd \Phi_5}\quad \dfrac{}{\Phi_5 \vdash \alpha'_2 \leqslant \alpha_3 \rhd \Phi_6}\quad \dots}{\Phi_3 \vdash \alpha'_1 \to \alpha'_2 \leq \alpha_2 \to \alpha_3 \rhd \Phi_7}\ \textsc{STypeConstr}}{\Phi_3 \vdash \alpha'_1 \to \alpha'_2 \leqslant \alpha_2 \to \alpha_3 \rhd \Phi_7}\ \textsc{SNewConstraint}}{\Phi_3 \vdash \alpha'_1 \to \alpha'_2 \leq \alpha_1 \rhd \Phi_7}\ \textsc{STransRight}}{\Phi_3 \vdash \alpha'_1 \to \alpha'_2 \leqslant \alpha_1 \rhd \Phi_7}\ \textsc{SNewConstraint}}{\Phi_2, \varnothing \vdash \lambda\ \mathtt{x}\ .\ \mathtt{e} : \alpha_1 \rhd \Phi_7}$$

  Nous commençons par construire un arbre d'inférence de $\Phi_2, \varnothing \vdash \mathtt{e}[\mathtt{x} \mapsto \mathtt{v}] : \alpha'_2 \rhd \Phi_8$ à partir de T_1 en y remplaçant les sous-arbres correspondant au typage des différentes

occurrences de $\mathtt{x}$ dans $\mathtt{e}$ par une adaptation de T_4 correspondant au typage de $\mathtt{v}$:



$$\Phi_2, \varnothing \vdash \mathtt{e}[\mathtt{x} \mapsto \mathtt{v}] : \alpha'_2 \vdash \Phi_8$$

Nous allons maintenant montrer comment sont construits T'_4, T''_4, etc., à partir de T_4.

Les sous-arbre de T_1 correspondant aux différentes occurrences de $\mathtt{x}$ dans $\mathtt{e}$ sont de la forme :

$$\textsc{Inst}\ \frac{\dfrac{\ldots}{\Phi_9 \vdash \alpha'_1 \leqslant \alpha' \rhd \Phi_{10}}}{\Phi_9, \{\ldots, (\mathtt{x} : \alpha'_1), \ldots\} \vdash \mathtt{x} : \alpha' \rhd \Phi_{10}}$$

Ce qui montre que Φ_{10} contient la contrainte $(\alpha'_1 \leqslant \alpha')$. De plus, le sous-arbre T_2 est une preuve de cohérence de la contrainte $(\alpha_2 \leqslant \alpha'_1)$ avec Φ_4. L'ensemble de contraintes Φ_5 contient donc la contrainte $(\alpha_2 \leqslant \alpha'_1)$. Comme aucune règle ne supprime de contrainte, l'ensemble de contraintes Φ_7 contient obligatoirement les contraintes $(\alpha_2 \leqslant \alpha'_1)$ et $(\alpha'_1 \leqslant \alpha')$. D'après le lemme 2, puisque Φ est saturé, l'ensemble de contraintes Φ_7 est saturé. Il contient donc en particulier la contrainte $(\alpha_2 \leqslant \alpha')$. Tous les sous-arbres de T_4 contiennent également cette contrainte, ce qui nous permet de déduire que, quelle que soit la structure de la valeur $\mathtt{v}$, le sous-arbre T_4 possède un sous-arbre T_0 de la forme :

$$\textsc{SNewConstraint}\ \dfrac{\textsc{STransLeft}\ \dfrac{\dfrac{\ldots}{\ldots}\quad\dfrac{\quad T_5\quad}{\Phi_{12} \vdash \tau_0^l \leqslant \alpha' \rhd \Phi_{13}}\quad\dfrac{\ldots}{\ldots}}{\Phi_{11} \vdash \tau_0^l \leq \alpha_2 \rhd \Phi_{14}}}{\Phi_{11} \vdash \tau_0^l \leqslant \alpha_2 \rhd \Phi_{14}}$$

où τ_0^l est un type défini par la structure de $\mathtt{v}$ (une flèche si $\mathtt{v}$ est un λ, une variable de type si $\mathtt{v}$ est une constante, etc.).

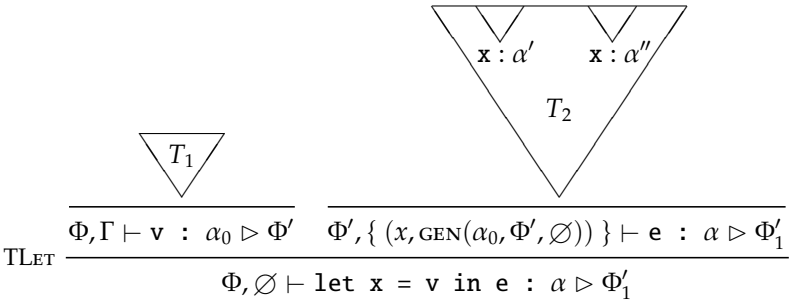
Comme la variable de type α'_1 apparaît libre dans les environnements de typage présents dans T_1, cela nous assure que α'_1 ne sera jamais généralisée dans T_1. Il n'y a donc aucun risque que des contraintes soient imposées sur une version renommée de α'_1 dans T_1 et donc ignorées dans T_4. Toutes les contraintes accumulées sur α'_1 dans T_1 sont donc prouvées compatibles avec τ_0^l dans T_4.

Au final, l'arbre d'inférence de $\Phi_2, \varnothing \vdash e[x \mapsto v] : \alpha'_2 \vdash \Phi_8$ est obtenu en remplaçant tous

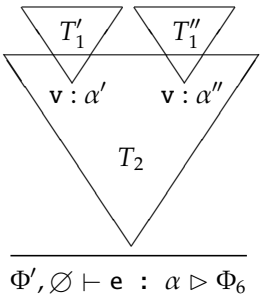
les sous-arbres de T_1 correspondant au typage des différentes occurrences de `x` dans `e` par T_4 dans lequel on a remplacé T_0 par T_5. Comme T_3 a prouvé que $\alpha'_2 \leqslant \alpha$ était compatible avec les autres contraintes, en saturant l'ensemble de contraintes, il a en particulier ajouté sur α toutes les contraintes accumulées sur α'_5 dans T_1. Nous en déduisons qu'il est possible de construire un arbre au dessus de $\Phi_2, \varnothing \vdash e[x \mapsto v] : \alpha \vdash \Phi'_8$ générant un ensemble de contraintes $\Phi'_8 \leqslant \Phi_8$.

Cette transformation d'arbre n'a ajouté aucune contrainte qui n'était pas déjà dans Φ'_1, nous en déduisons que $\Phi'_8 \leqslant \Phi'_1$. De plus $\Phi_2 = \Phi \wedge \alpha_1 \leqslant \alpha_2 \to \alpha \wedge \alpha_3 \leqslant \alpha$, donc $\Phi \leqslant \Phi_2$. D'après le lemme 4, nous en déduisons que $\texttt{e}[\texttt{x} \mapsto \texttt{v}]$ est typable dans Φ et génère un ensemble de contraintes Φ'_2 vérifiant $\Phi'_2 \leqslant \Phi'_8$. Comme $\Phi'_8 \leqslant \Phi'_1$, par transitivité de $(\leqslant)$, les ensembles de contraintes Φ'_1 et Φ'_2 vérifient bien $\Phi'_2 \leqslant \Phi'_1$. □

- Cas « `let x = v in e` $\longrightarrow$ `e[x` $\mapsto$ `v]` » :
  L'arbre d'inférence de $\Phi, \varnothing \vdash \texttt{let x = v in e} : \alpha \rhd \Phi'_1$ est obligatoirement de la forme :



Nous construisons alors l'arbre d'inférence de $\Phi', \varnothing \vdash \texttt{e} : \alpha \rhd \Phi_6$ en remplaçant dans T_2 les sous-arbres correspondant au typage des différentes occurrences de `x` dans `e` par une adaptation de T_1 :



Ces sous-arbres sont tous de la forme :

$$\textsc{TVar}\ \dfrac{\textsc{Inst}\ \dfrac{\dfrac{\overset{T_3}{\triangledown}}{\Phi_3 \vdash \alpha_0[\alpha_i \mapsto \alpha'_i]_{i=1}^n \leqslant \alpha' \rhd \Phi_4} \quad \dfrac{\overset{T_4}{\triangledown}}{\Phi_4 \vdash \Phi'[\alpha_i \mapsto \alpha'_i]_{i=1}^n \rhd \Phi_5}}{\Phi_3 \vdash \textsc{gen}(\alpha_0, \Phi', \varnothing) \leqslant \alpha' \rhd \Phi_5}}{\Phi_3, \{\ \ldots, (\mathtt{x}, \textsc{gen}(\alpha_0, \Phi', \varnothing)), \ldots\} \vdash \mathtt{x} : \alpha' \rhd \Phi_5}$$

Nous les remplaçons par T_1 dans lequel les instances des noeuds de typage et d'instanciation sont identiques (car imposés par la structure de v). De plus, les sous-arbres T_3 et T_4 ont montré que les contraintes accumulées sur α_0 lors de la construction de T_1 sont compatibles avec les contraintes accumulées sur α' dans T_2, ce qui montre que les sous-arbres de saturation sont constructibles et que $\Phi_6 \leqslant \Phi'_1$.

Enfin, nous avons $\Phi \leqslant \Phi'$ et $\Phi', \varnothing \vdash \mathtt{e} : \alpha \rhd \Phi_6$, donc d'après le lemme 4, l'expression e est typable dans $(\Phi, \varnothing)$ et la construction de l'arbre d'inférence au dessus de $\Phi, \varnothing \vdash \mathtt{e} : \alpha \rhd \Phi'_2$ génère un ensemble de contraintes Φ'_2 vérifiant $\Phi'_2 \leqslant \Phi_6$. Comme $\Phi_6 \leqslant \Phi'_1$, par transitivité de $(\leqslant)$, nous obtenons bien $\Phi'_2 \leqslant \Phi'_1$.

- Cas « `if true then` $\mathtt{e}_1$ `else` $\mathtt{e}_2 \longrightarrow \mathtt{e}_1$ » :
  L'arbre d'inférence de $\Phi, \varnothing \vdash$ `if true then` $\mathtt{e}_1$ `else` $\mathtt{e}_2 : \alpha \rhd \Phi'_1$ est de la forme :

$$\textsc{TIf}\ \dfrac{\dfrac{\overset{}{\triangledown}}{\Phi \vdash \alpha' \leqslant \mathtt{bool} \rhd \Phi_1} \quad \dfrac{\overset{}{\triangledown}}{\Phi_1 \vdash \mathtt{true} : \alpha' \rhd \Phi_2} \quad \dfrac{\overset{T}{\triangledown}}{\Phi_2, \varnothing \vdash \mathtt{e}_1 : \alpha \rhd \Phi_3} \quad \dfrac{\overset{}{\triangledown}}{\Phi_3, \varnothing \vdash \mathtt{e}_2 : \alpha \rhd \Phi'_1}}{\Phi, \varnothing \vdash \mathtt{if\ true\ then}\ \mathtt{e}_1\ \mathtt{else}\ \mathtt{e}_2 : \alpha \rhd \Phi'_1}$$

  avec $\Phi \leqslant \Phi_2$ et $\Phi_3 \leqslant \Phi'_1$.

  Le sous-arbre T montre $\Phi_2, \varnothing \vdash \mathtt{e}_1 : \alpha \rhd \Phi_3$. Puisque $\Phi \leqslant \Phi_2$, une simple application du lemme 4 nous montre qu'il existe un Φ'_2 tel que $\Phi, \varnothing \vdash \mathtt{e}_1 : \alpha \rhd \Phi'_2$ avec $\Phi'_2 \leqslant \Phi_3$. Comme $\Phi_3 \leqslant \Phi'_1$, par transitivité de $(\leqslant)$, nous obtenons bien $\Phi'_2 \leqslant \Phi'_1$.

- Cas « `if false then` $\mathtt{e}_1$ `else` $\mathtt{e}_2 \longrightarrow \mathtt{e}_2$ » :
  La démonstration est très similaire à celle du cas précédent.

- Cas « `match K v with ... ‖ K x → e ‖ ... ⟶ e[x ↦ v]` » :
  L'arbre d'inférence de

$$\Phi, \varnothing \vdash \texttt{match K v with ... ‖ K x → e ‖ ... :} \ \alpha \rhd \Phi_1'$$

  est de la forme :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Phi_2 \vdash \alpha_0 \leqslant \alpha_1 \rhd \Phi_3}\,T_3}{\Phi_2 \vdash \mathtt{K}\,\alpha_0 \leq \{ \ldots \parallel \mathtt{K}\,\alpha_1 \parallel \ldots \} \rhd \Phi_3}\,\textsc{SVariantMatch}}{\Phi_2 \vdash \mathtt{K}\,\alpha_0 \leqslant \{ \ldots \parallel \mathtt{K}\,\alpha_1 \parallel \ldots \} \rhd \Phi_3}\,\textsc{SNewConstraint}}{\Phi_2 \vdash \mathtt{K}\,\alpha_0 \leq \alpha_e \rhd \Phi_3}\,\textsc{STransRight}}{\Phi_1, \varnothing \vdash \mathtt{v} : \alpha_0 \rhd \Phi_2 \qquad \Phi_2 \vdash \mathtt{K}\,\alpha_0 \leqslant \alpha_e \rhd \Phi_3}\,\textsc{SNewConstraint}}{\Phi_1, \varnothing \vdash \mathtt{K\ v} : \alpha_e \rhd \Phi_3}\,\textsc{TConstr}$$

$$\dfrac{\dfrac{\dots \qquad \dfrac{\mathtt{x}:\alpha' \quad \mathtt{x}:\alpha''}{\Phi_3, \{ (\mathtt{x}, \alpha_1) \} \vdash \mathtt{e} : \alpha \rhd \Phi_4}\,T_4 \qquad \dots}{\dots \qquad\qquad\qquad\qquad \dots}}{\Phi, \varnothing \vdash \texttt{match K v with ... ‖ K x → e ‖ ... :} \ \alpha \rhd \Phi_1'}\,\textsc{TMatch}$$

  Nous nous retrouvons alors dans une situation très similaire à celle du troisième cas (« `(λ x . e) v ⟶ e[x ↦ v]` »). De la même manière, nous construisons un arbre d'inférence au dessus de $\Phi_3, \varnothing \vdash \mathtt{e[x \mapsto v]} : \alpha \rhd \Phi_5$ en remplaçant les sous-arbres correspondant au typage des différentes occurrences de `x` dans `e` par une adaptation de T_2. Ceci est possible puisque `x` est lié à α_1 dans l'environnement de typage utilisé pour construire T_4, le sous-arbre T_2 représente une preuve de typage de $(\mathtt{v} : \alpha_0)$ et les variables α_0 et α_1 sont reliées par la relation $\alpha_0 \leqslant \alpha_1$ grâce à T_3.

  Enfin, tout comme dans le troisième cas, nous utilisons le lemme 4 pour montrer qu'il existe Φ_2' tel que $\Phi, \varnothing \vdash \mathtt{e[x \mapsto v]} : \alpha \rhd \Phi_2'$ avec $\Phi_2' \leqslant \Phi_5 \leqslant \Phi_1'$.

- Cas « `match K v with ... ‖ K x → e ‖ ... ‖ x_d → e_d ⟶ e[x ↦ v]` » :

La démonstration est très similaire à celle du cas précédent.

- Cas « $\texttt{match v with K}_1\ \texttt{x}_1 \to \texttt{e}_1 \parallel \ldots \parallel \texttt{K}_n\ \texttt{x}_n \to \texttt{e}_n \parallel \texttt{x}_d \to \texttt{e}_d \longrightarrow \texttt{e}_d[\texttt{x}_d \mapsto \texttt{v}]$ »
  lorsque v n'est pas de la forme $(\texttt{K}_i\ \_)$ pour $i \in [\,1;\ n\,]$:
  L'arbre d'inférence de

$$\Phi, \varnothing \vdash \texttt{match v with K}_1\ \texttt{x}_1 \to \texttt{e}_1 \parallel \ \ldots \ \parallel \texttt{K}_n\ \texttt{x}_n \to \texttt{e}_n \parallel \ \texttt{x}_d \to \texttt{e}_d : \alpha \rhd \Phi'_1$$

  est de la forme :

$$\textsc{TMatchDefault}\ \dfrac{\dfrac{T_2}{\Phi_1, \varnothing \vdash \texttt{v} : \alpha_e \rhd \Phi_2} \quad \dfrac{\ldots}{\ldots} \quad \dfrac{\overset{\texttt{x}_d : \alpha' \qquad \texttt{x}_d : \alpha''}{T_3}}{\Phi_2, \{\ (\texttt{x}_d, \alpha_d)\ \} \vdash \texttt{e}_d : \alpha \rhd \Phi'_1}}{\Phi, \varnothing \vdash \texttt{match v with K}_1\ \texttt{x}_1 \to \texttt{e}_1 \parallel \ \ldots \ \parallel \texttt{K}_n\ \texttt{x}_n \to \texttt{e}_n \parallel \ \texttt{x}_d \to \texttt{e}_d : \alpha \rhd \Phi'_1}$$

  Comme v n'est pas de la forme $(\texttt{K}_i\ \_)$, le sous-arbre T_2 génère une contrainte entre un type construit τ_0^l de la forme $(\tau_0^l \leqslant \alpha_e)$ avec τ_0^l qui n'est pas de la forme $\texttt{K}_i\ \_$ pour $i \in [\,1;\ n]$. Comme Φ_1 contient la contrainte $(\alpha_e \leqslant \{\ \texttt{K}_1\ \alpha_1 \parallel \ \ldots \ \parallel \texttt{K}_n\ \alpha_n \parallel \ \alpha_d\ \})$, un sous-arbre de T_2 prouve $(\tau_0^l \leqslant \{\ \texttt{K}_1\ \alpha_1 \parallel \ \ldots \ \parallel \texttt{K}_n\ \alpha_n \parallel \ \alpha_d\ \})$ et comme τ_0^l n'est pas de la forme $\texttt{K}_{\ i}$ pour $i \in [\,1;\ n\,]$, une instance de la règle SVariantDefault ou SConstrDefault impose la présence de $\tau_0^l \leqslant \alpha_d$ dans Φ_2.
  On se retrouve alors dans un cas très similaire au précédent. On montre qu'il est possible de construire l'arbre d'inférence de $\Phi_2, \varnothing \vdash \texttt{e}_d[\texttt{x}_d \mapsto \texttt{v}] : \alpha \rhd \Phi_3$ en remplaçant dans T_3 les sous-arbres correspondant au typage des occurrences libres de x dans e par une adaptation de T_2. Enfin, le lemme 4 nous montre qu'il existe Φ'_2 tel que $\Phi, \varnothing \vdash \texttt{e}_d[\texttt{x}_d \mapsto \texttt{v}] : \alpha \rhd \Phi'_2$ avec $\Phi'_2 \leqslant \Phi'_1$.

Lemme 6 (Réduction du sujet par ($\longmapsto$))**.**

Soient :

- e_1 *et* e_2 *deux expressions vérifiant* $\mathsf{e}_1 \longmapsto \mathsf{e}_2$
- α *une variable de type*
- Φ *un ensemble de contraintes valide et saturé*
- Φ'_1 *l'ensemble de contraintes obtenu par typage de* $\mathsf{e}_1 : \alpha$ *dans* $(\Phi, \varnothing)$ *en dérivant l'arbre d'inférence au dessus de* $\Phi, \varnothing \vdash \mathsf{e}_1 : \alpha \rhd \Phi'_1$.

Alors le typage de $\mathsf{e}_2 : \alpha$ *dans* $(\Phi, \varnothing)$ *réussit et l'ensemble de contraintes* Φ'_2 *obtenu en dérivant l'arbre d'inférence au dessus de* $\Phi, \varnothing \vdash \mathsf{e}_2 : \alpha \rhd \Phi'_2$ *vérifie* $\Phi'_2 \leqslant \Phi'_1$.

Démonstration (Lemme 6) :

. . . par induction et analyse de cas sur la structure du contexte d'évaluation `E` :

- Cas « `[]` » :
  Conséquence directe du lemme 5.

- Cas « `E e` » :
  Nous souhaitons montrer qu'il est possible de construire l'arbre d'inférence de $\Phi, \varnothing \vdash \mathsf{e}_2\ \mathsf{e} : \alpha \rhd \Phi'_2$ grâce à l'arbre d'inférence de $\Phi, \varnothing \vdash \mathsf{e}_1\ \mathsf{e} : \alpha \rhd \Phi'_1$ sachant que $\mathsf{e}_1 \longmapsto \mathsf{e}_2$. D'après la structure de l'expression (une application), la règle de typage appliquée à la racine de ces arbres est obligatoirement TApp :

$$\textsc{TApp}\ \dfrac{\dfrac{\overset{T_1}{\triangledown}}{\Phi \vdash \alpha_1 \leqslant \alpha_2 \to \alpha_3 \rhd \Phi_1} \quad \dfrac{\overset{T_2}{\triangledown}}{\Phi_1 \vdash \alpha_3 \leqslant \alpha \rhd \Phi_2} \qquad \dfrac{\dfrac{\overset{T_3}{\triangledown}}{\Phi_2, \varnothing \vdash \mathsf{e}_1 : \alpha_1 \rhd \Phi_3} \quad \dfrac{\overset{T_4}{\triangledown}}{\Phi_3, \varnothing \vdash \mathsf{e} : \alpha_2 \rhd \Phi'_1}}{\Phi, \varnothing \vdash \mathsf{e}_1\ \mathsf{e} : \alpha \rhd \Phi'_1}}$$

  Les sous-arbres T_1 et T_2 sont inchangés. L'hypothèse d'induction nous montre que le sous-arbre remplaçant T_3 peut être construit en générant un ensemble de contraintes plus petit. Et enfin, le lemme 4 nous montre que le sous-arbre remplaçant T_4 peut être construit en générant un ensemble de contraintes Φ'_2 plus petit que Φ'_1.

- Cas « `v E` » :
  La structure est similaire à celle du cas précédents. Les sous-arbres T_1, T_2 et T_3 restent inchangés. On utilise l'hypothèse d'induction pour montrer qu'il est possible de

remplacer T_4 en générant un ensemble de contraintes plus petit.

- Cas « $\mathtt{p}^1$ `E` » :
  La situation est similaire mis à part que la règle TAppPrim1 est utilisée à la place de TApp. La structure de l'arbre initial est :

$$\textsc{TApplyPrim1}\ \frac{\dfrac{\overset{T_1}{\triangledown}}{\Phi \vdash T(\mathtt{p}^1) \leqslant \alpha_1 \to \alpha_2 \rhd \Phi_1} \qquad \dfrac{\dfrac{\overset{T_2}{\triangledown}}{\Phi_1 \vdash \alpha_2 \leqslant \alpha \rhd \Phi_2} \qquad \dfrac{\overset{T_3}{\triangledown}}{\Phi_2, \varnothing \vdash \mathtt{e}_1 : \alpha_1 \rhd \Phi_1'}}{}}{\Phi, \varnothing \vdash \mathtt{p}^1\ \mathtt{e}_1 : \alpha \rhd \Phi_1'}$$

  Les sous-arbres T_1 et T_2 sont inchangés, on utilise l'hypothèse d'induction pour montrer qu'il est possible de remplacer T_3 en générant un ensemble de contraintes Φ_2' plus petit que Φ_1'.

- Cas « $\mathtt{p}^2$ `E e` » :
  Similaire au cas précédent.

- Cas « $\mathtt{P}^2$ `v E` » :
  Similaire au cas précédent.

- Cas « `(E, e)` » :
  Nous souhaitons montrer qu'il est possible de construire l'arbre d'inférence de $\Phi, \varnothing \vdash (\mathtt{e}_2,\ \mathtt{e}) : \alpha \rhd \Phi_2'$ grâce à l'arbre d'inférence de $\Phi, \varnothing \vdash (\mathtt{e}_1,\ \mathtt{e}) : \alpha \rhd \Phi_1'$ sachant que $\mathtt{e}_1 \longmapsto \mathtt{e}_2$. Ce dernier arbre est de la forme :

$$\textsc{TPair}\ \frac{\overset{T_1}{\triangledown} \qquad \overset{T_2}{\triangledown} \qquad \overset{T_3}{\triangledown}}{\Phi, \varnothing \vdash \mathtt{e}_1 : \alpha_1 \rhd \Phi_1 \qquad \Phi_1, \varnothing \vdash \mathtt{e} : \alpha_2 \rhd \Phi_2 \qquad \Phi_2 \vdash \alpha_1 \times \alpha_2 \leqslant \alpha \rhd \Phi_1'}{\Phi, \varnothing \vdash (\mathtt{e}_1,\ \mathtt{e}) : \alpha \rhd \Phi_1'}$$

  Comme dans les cas précédent, on utilise l'hypothèse d'induction pour montrer qu'il est possible de remplacer T_1 par un sous-arbre générant un ensemble de contraintes plus petit, le lemme 4 pour le remplacement du sous-arbre T_2 et le lemme 3 pour le remplacement du sous-arbre T_3.

- Cas « `(v, E)` » :
  Similaire au cas précédent.

- Cas « `K E` » :
  Similaire au cas précédent.

- Cas « `let x = E in e` » :
  Nous souhaitons montrer qu'il est possible de construire l'arbre d'inférence de $\Phi, \varnothing \vdash \texttt{let x = } \mathtt{e}_2 \texttt{ in e} : \alpha \rhd \Phi'_2$ grâce à l'arbre d'inférence de $\Phi, \varnothing \vdash \texttt{let x = } \mathtt{e}_1 \texttt{ in e} : \alpha \rhd \Phi'_1$ sachant que $\mathtt{e}_1 \longmapsto \mathtt{e}_2$. Ce dernier arbre est de la forme :

$$\textsc{TLet}\ \frac{\dfrac{\overset{T_1}{\triangledown}}{\Phi, \varnothing \vdash \mathtt{e}_1 : \alpha' \rhd \Phi_1} \quad \dfrac{\overset{T_2}{\triangledown}}{\Phi_1, \{\ (\mathtt{x}, \textsc{gen}(\alpha', \Phi_1, \varnothing))\ \} \vdash \mathtt{e} : \alpha \rhd \Phi'_1}}{\Phi, \varnothing \vdash \texttt{let x} = \mathtt{e}_1 \texttt{ in e} : \alpha \rhd \Phi'_1}$$

  La structure de l'arbre d'inférence de $\Phi, \varnothing \vdash \texttt{let x = } \mathtt{e}_2 \texttt{ in e} : \alpha \rhd \Phi'_2$ que nous allons construire est similaire :

$$\textsc{TLet}\ \frac{\dfrac{\overset{T'_1}{\triangledown}}{\Phi, \varnothing \vdash \mathtt{e}_2 : \alpha' \rhd \Phi_2} \quad \dfrac{\overset{T'_2}{\triangledown}}{\Phi_2, \{\ (\mathtt{x}, \textsc{gen}(\alpha', \Phi_1, \varnothing))\ \} \vdash \mathtt{e} : \alpha \rhd \Phi'_1}}{\Phi, \varnothing \vdash \texttt{let x} = \mathtt{e}_2 \texttt{ in e} : \alpha \rhd \Phi'_1}$$

  L'hypothèse d'induction nous montre qu'il est effectivement possible de construire T'_1, et que Φ_2 vérifie $\Phi_2 \leqslant \Phi_1$. Nous en déduisons la relation suivante entre schémas de type :

$$\textsc{gen}(\alpha', \Phi_2, \varnothing) \leqslant \textsc{gen}(\alpha', \Phi_1, \varnothing)$$

  et donc la relation suivante entre environnements de typage :

$$\{\ (x, \textsc{gen}(\alpha', \Phi_2, \varnothing))\ \} \leqslant \{\ (x, \textsc{gen}(\alpha', \Phi_1, \varnothing))\ \}$$

  Grâce au lemme 4, nous concluons que T'_2 est effectivement constructible, et que l'ensemble de contraintes Φ'_2 généré vérifie bien $\Phi'_2 \leqslant \Phi'_1$.

- Cas « `if E then` $\mathtt{e}_1$ `else` $\mathtt{e}_2$ » :
  Nous souhaitons montrer qu'il est possible de construire l'arbre d'inférence de $\Phi, \varnothing \vdash \texttt{if } \mathtt{e}_2 \texttt{ then } \mathtt{e}_3 \texttt{ else } \mathtt{e}_4 : \alpha \rhd \Phi'_1$ grâce à l'arbre d'inférence de $\Phi, \varnothing \vdash \texttt{if } \mathtt{e}_1 \texttt{ then } \mathtt{e}_3 \texttt{ else } \mathtt{e}_4 : \alpha \rhd \Phi'_1$ sachant que $\mathtt{e}_1 \longmapsto \mathtt{e}_2$. Ce dernier arbre est de la

forme :

$$\textsc{TIf}\ \dfrac{\dfrac{\overset{T_1}{\triangledown}}{\Phi \vdash \alpha' \leqslant \texttt{bool} \rhd \Phi_1} \quad \dfrac{\overset{T_2}{\triangledown}}{\Phi_1, \varnothing \vdash \texttt{e}_1 : \alpha' \rhd \Phi_2} \quad \dfrac{\overset{T_3}{\triangledown}}{\Phi_2, \varnothing \vdash \texttt{e}_3 : \alpha \rhd \Phi_3} \quad \dfrac{\overset{T_4}{\triangledown}}{\Phi_3, \varnothing \vdash \texttt{e}_4 : \alpha \rhd \Phi_4}}{\Phi, \varnothing \vdash \texttt{if}\ \texttt{e}_1\ \texttt{then}\ \texttt{e}_3\ \texttt{else}\ \texttt{e}_4 : \alpha \rhd \Phi_1'}$$

Comme précédemment, le sous-arbre T_1 ne change pas, il suffit d'appliquer l'hypothèse d'induction pour montrer qu'il est possible de remplacer T_2 par un sous-arbre générant un ensemble de contraintes plus petit, et le lemme 4 deux fois pour T_3 et T_4.

- Cas « `match E with` $\texttt{K}_1\ \texttt{x}_1 \to \texttt{e}_1 \parallel \ldots \parallel \texttt{K}_n\ \texttt{x}_n \to \texttt{e}_n$ » :
  Nous souhaitons montrer qu'il est possible de construire l'arbre d'inférence de `match` $\texttt{e}_1$ `with` $\texttt{K}_1\ \texttt{x}_1 \to \texttt{e}_1' \parallel \ldots \parallel \texttt{K}_n\ \texttt{x}_n \to \texttt{e}_n'$ grâce à l'arbre d'inférence de `match` $\texttt{e}_2$ `with` $\texttt{K}_1\ \texttt{x}_1 \to \texttt{e}_1' \parallel \ldots \parallel \texttt{K}_n\ \texttt{x}_n \to \texttt{e}_n'$ sachant que $\texttt{e}_1 \longmapsto \texttt{e}_2$. Ce dernier arbre est de la forme :

$$\textsc{TMatch}\ \dfrac{\dfrac{\overset{T_1}{\triangledown}}{\Phi \vdash \alpha_e \leqslant \{\, \texttt{K}_1\ \alpha_1 \parallel \ldots \parallel \texttt{K}_n\ \alpha_n \,\} \rhd \Phi_0} \quad \dfrac{\overset{T_2}{\triangledown}}{\Phi_0, \varnothing \vdash \texttt{e}_1 : \alpha_e \rhd \Phi_1} \quad \dfrac{\triangledown}{\Phi_1, \varnothing \oplus (\texttt{x}_1,\ \alpha_1) \vdash \texttt{e}_1' : \alpha \rhd \Phi_2} \quad \ldots \quad \dfrac{\triangledown}{\Phi_n, \varnothing \oplus (\texttt{x}_n,\ \alpha_n) \vdash \texttt{e}_n' : \alpha \rhd \Phi_1'}}{\Phi, \varnothing \vdash \texttt{match}\ \texttt{e}_1\ \texttt{with}\ \texttt{K}_1\ \texttt{x}_1 \to \texttt{e}_1 \parallel \ldots \parallel \texttt{K}_n\ \texttt{x}_n \to \texttt{e}_n : \alpha \rhd \Phi_1'}$$

  Comme précédemment, le sous-arbre T_1 est inchangé. On utilise l'hypothèse d'induction pour montrer qu'il est possible de remplacer T_2 par un sous-arbre générant un ensemble de contraintes plus petit, et le lemme 4 pour les autres sous-arbres.

- Cas « `match E with` $\texttt{K}_1\ \texttt{x}_1 \to \texttt{e}_1 \parallel \ldots \parallel \texttt{K}_n\ \texttt{x}_n \to \texttt{e}_n \parallel \texttt{x}_d \to \texttt{e}_d$ » :
  Similaire au cas précédent.

Lemme 7 (Réduction du sujet par $(\longmapsto\!\!\!\!\!\rightarrow\!\!\!\!\!\rightarrow)$)**.**

Soient :

- e_1 *et* e_2 *deux expressions vérifiant* $e_1 \longmapsto\!\!\!\!\!\rightarrow\!\!\!\!\!\rightarrow e_2$.
- α *une variable de type*
- Φ *un ensemble de contraintes valide et saturé*
- Φ'_1 *l'ensemble de contraintes obtenu par typage de* $e_1 : \alpha$ *dans* $(\Phi, \varnothing)$ *en dérivant l'arbre d'inférence au dessus de* $\Phi, \varnothing \vdash e_1 : \alpha \rhd \Phi'_1$.

Alors le typage de $e_2 : \alpha$ *dans* $(\Phi, \varnothing)$ *réussit et l'ensemble de contraintes* Φ'_2 *obtenu en dérivant l'arbre d'inférence au dessus de* $\Phi, \varnothing \vdash e_2 : \alpha \rhd \Phi'_2$ *vérifie* $\Phi'_2 \leqslant \Phi'_1$.

Démonstration (Lemme 7) :

... par induction et analyse de cas sur la définition de $(\longmapsto\!\!\!\!\!\rightarrow\!\!\!\!\!\rightarrow)$:

- Si $e_1 \longmapsto\!\!\!\!\!\rightarrow\!\!\!\!\!\rightarrow e_2$ parce que $e_1 = e_2$:
  Trivial.

- Si $e_1 \longmapsto\!\!\!\!\!\rightarrow\!\!\!\!\!\rightarrow e_2$ parce que $e_1 \longmapsto e_2$:
  Simple conséquence du lemme 6.

- Si $e_1 \longmapsto\!\!\!\!\!\rightarrow\!\!\!\!\!\rightarrow e_2$ parce qu'il existe une expression e telle que $e_1 \longmapsto\!\!\!\!\!\rightarrow\!\!\!\!\!\rightarrow e$ et $e \longmapsto e_2$:
  En appliquant l'hypothèse d'induction, nous montrons que le typage de $e : \alpha$ réussit dans $(\Phi, \varnothing)$ et produit un ensemble de contraintes Φ' vérifiant $\Phi' \leqslant \Phi'_1$:

  $$\Phi, \varnothing \vdash e : \alpha \rhd \Phi'$$

  En appliquant le lemme 6, nous montrons que le typage de $e_2 : \alpha$ dans $(\Phi, \varnothing)$ réussit et produit un ensemble de contraintes que nous nommons Φ'_2 vérifiant $\Phi'_2 \leqslant \Phi'$:

  $$\Phi, \varnothing \vdash e_2 : \alpha \rhd \Phi'_2$$

  Comme la relation $(\leqslant)$ est transitive, nous obtenons bien $\Phi'_2 \leqslant \Phi'_1$.

Lemme 8 (Les expressions bloquées ne sont pas typables)**.**

Soit e_b *une expression bloquée. Il n'existe aucun schéma de type* σ *tel que* $e_b : \sigma$.

Démonstration (Lemme 8) :

... par induction sur la structure des expressions et analyse de cas sur les différentes formes d'expressions bloquées. En réalité, l'analyse de chacun des cas est trivial. Le principal intérêt de cette démonstration est de répertorier toutes les formes d'expressions bloquées.

L'expression $\mathtt{e}_b$ étant bloquée, elle est obligatoirement de l'une des formes suivantes :

- Cas « $\mathtt{e}_b = \mathtt{x}$ » :
  L'environnement de typage étant vide, la règle TVar ne peut pas être appliquée et la variable $\mathtt{x}$ n'est pas typable et on obtient le message classique « unbound variable $\mathtt{x}$ ».
- Cas « $\mathtt{e}_b = \mathtt{c}\ \ \mathtt{v}$ » :
  S'il était constructible complètement, de par la structure de l'expression, l'arbre d'inférence serait de la forme :

$$\dfrac{\dfrac{\dfrac{}{\varnothing \vdash \alpha_1 \leqslant \alpha_2 \to \alpha_3 \rhd \Phi'}\ T_1 \qquad \dfrac{}{\Phi' \vdash \alpha_3 \leqslant \alpha \rhd \Phi''}\ T_2}{\Phi'', \varnothing \vdash \mathtt{c} : \alpha_1 \rhd \Phi'''}\ \textsc{TConst} \qquad \dfrac{}{\Phi''' \vdash T(c) \leqslant \alpha_1 \rhd \Phi'''}\ T_3 \qquad \dfrac{}{\Phi''', \varnothing \vdash \mathtt{v} : \alpha_2 \rhd \Phi''''}\ T_4}{\varnothing, \varnothing \vdash \mathtt{c}\ \mathtt{v} : \alpha \rhd \Phi''''}\ \textsc{TApp}$$

  On fait ici l'hypothèse que la fonction T est correctement construite, et que puisqu'aucune constante n'est fonctionnelle, l'instanciation de $T(\mathtt{c})$ génère obligatoirement un type construit τ^l différent de $(\to)$ et vérifiant $(\tau^l \leqslant \alpha_1)$. Après application de la transitivité de $(\leqslant)$ via la règle STransRight dans T_2, on aboutirait à la contrainte $(\tau^l \leq \alpha_2 \to \alpha_3)$. Comme τ^l est un type construit différent de $(\to)$, aucune règle de typage ne s'appliquerait pour saturer la contrainte $(\tau^l \leq \alpha_2 \to \alpha_3)$ et la construction de l'arbre d'inférence s'arrêterait bien sur une erreur.
- Cas « $\mathtt{e}_b = (\mathtt{K}\ \ \mathtt{v}_1)\ \ \mathtt{v}_2$ » :
  L'arbre d'inférence, s'il était constructible complètement, serait de la forme :

$$\dfrac{\dfrac{\dfrac{}{\varnothing \vdash \alpha_1 \leqslant \alpha_2 \to \alpha_3 \rhd \Phi'}\ T_1 \qquad \dfrac{}{\Phi' \vdash \alpha_3 \leqslant \alpha \rhd \Phi''}\ T_2}{\Phi'', \varnothing \vdash \mathtt{K}\ \mathtt{v}_1 : \alpha_1 \rhd \Phi'''}\ \textsc{TConstr} \qquad \dfrac{}{\Phi''' \vdash \mathtt{v}_2 : \alpha_2 \rhd \Phi''''}\ T_4}{\varnothing, \varnothing \vdash (\mathtt{K}\ \mathtt{v}_1)\ \mathtt{v}_2 : \alpha \rhd \Phi''''}\ \textsc{TApp} \qquad \dfrac{}{\Phi''', \varnothing \vdash \mathtt{v}_2 : \alpha_2 \rhd \Phi''''}\ T_3$$

  L'application de la règle TConstr dans T_3 engendrerait alors une contrainte de la forme $(\mathtt{K}\ \alpha \leqslant \alpha_1)$. Après application de la transitivité, on aboutirait à la contrainte $(\mathtt{K}\ \alpha \leqslant \alpha_2 \to \alpha_3)$ sur laquelle aucune règle ne s'applique.

- Cas « $\mathtt{e}_b = (\mathtt{v}_1, \mathtt{v}_2)\ \mathtt{v}_3$ » : Comme précédemment mais avec la règle TPair, on aboutirait à une contrainte de la forme $(\alpha_3 \times \alpha_4 \leqslant \alpha_2 \to \alpha)$ sur laquelle aucune règle de saturation ne s'applique.

- Cas « $\mathtt{e}_b = \mathtt{p}^1\ \mathtt{v}$ avec $\delta_1(\mathtt{p}^1, \mathtt{v})$ non défini » :
  S'il était constructible, l'arbre d'inférence serait de la forme :

$$\textsc{TApplyPrim1}\ \frac{\dfrac{\overset{T_1}{\triangledown}}{\varnothing \vdash T(\mathtt{p}^1) \leqslant \alpha_1 \to \alpha_2 \rhd \Phi'} \qquad \dfrac{\dfrac{\overset{T_2}{\triangledown}}{\Phi' \vdash \alpha_2 \leqslant \alpha \rhd \Phi''} \qquad \dfrac{\overset{T_3}{\triangledown}}{\Phi'', \varnothing \vdash \mathtt{v} : \alpha_1 \rhd \Phi'''}}{\varnothing, \varnothing \vdash \mathtt{p}^1\ \mathtt{v} : \alpha \rhd \Phi''}}$$

  Il s'agit ici d'une hypothèse sur la compatibilité de T et de δ_1. En l'occurrence, la fonction T doit être définie de telle sorte que les contraintes engendrées par $(T(\mathtt{p}^1) \leqslant \alpha_1 \to \alpha_2)$ doivent contraindre suffisamment α_1 pour que si le typage de $(\mathtt{v} : \alpha_1)$ réussit, alors $\delta_1(\mathtt{p}^1, \mathtt{v})$ est défini.

- Cas « $\mathtt{e}_b = \mathtt{p}^2\ \mathtt{v}_1\ \mathtt{v}_2$ avec $\delta_2(\mathtt{p}^2, \mathtt{v}_1, \mathtt{v}_2)$ non défini » :
  Similaire au cas précédent, mais avec δ_2.

- Cas « $\mathtt{e}_b = \mathtt{if}\ \lambda\,\mathtt{x}\,.\,\mathtt{e}\ \mathtt{then}\ \mathtt{e}_1\ \mathtt{else}\ \mathtt{e}_2$ » :
  Si l'arbre d'inférence était constructible, il serait de la forme :

$$\textsc{TIf}\ \frac{\dfrac{\overset{T_1}{\triangledown}}{\varnothing \vdash \alpha' \leqslant \mathtt{bool} \rhd \Phi'} \qquad \textsc{TLambda}\ \dfrac{\dfrac{\dots}{\dots} \quad \dfrac{\overset{T_2}{\triangledown}}{\Phi''' \vdash \alpha_1 \to \alpha_2 \leqslant \alpha' \rhd \Phi''''}}{\Phi', \varnothing \vdash \lambda \mathtt{x.e} : \alpha' \rhd \Phi''} \quad \dfrac{\dots}{\dots} \quad \dfrac{\dots}{\dots}}{\varnothing, \varnothing \vdash \mathtt{if}\ \lambda \mathtt{x.e}\ \mathtt{then}\ \mathtt{e}_1\ \mathtt{else}\ \mathtt{e}_2 \rhd \Phi'''''}$$

  Après application de la transitivité de $(\leqslant)$ dans T_2 par la règle STransRight, on aboutirait obligatoirement à la contrainte $(\alpha_1 \to \alpha_2 \leqslant \mathtt{bool})$ sur laquelle aucune règle de saturation ne s'applique.

- Cas « $\mathtt{e}_b = \mathtt{if}\ (\mathtt{v}_1, \mathtt{v}_2)\ \mathtt{then}\ \mathtt{e}_1\ \mathtt{else}\ \mathtt{e}_2$ » :
  Ce cas est similaire au précédent. L'application de la règle TPair engendrerait une contrainte de la forme $(\alpha_1 \times \alpha_2 \leqslant \mathtt{bool})$ sur laquelle aucune règle ne s'applique.

- Cas « $\mathtt{e}_b = \mathtt{if}\ \mathtt{K}\ \mathtt{v}\ \mathtt{then}\ \mathtt{e}_1\ \mathtt{else}\ \mathtt{e}_2$ » :
  Ce cas est similaire au précédent. L'application de la règle TConstr engendrerait une contrainte de la forme $(\mathtt{K}\,\alpha_1 \leqslant \mathtt{bool})$ sur laquelle aucune règle ne s'applique.

- Cas « $\mathtt{e}_b = \mathtt{if\ c\ then\ e}_1\ \mathtt{else\ e}_2$ avec $\mathtt{c} \notin \{\ \mathtt{true},\ \mathtt{false}\ \}$ » :
  Il s'agit une nouvelle fois d'une hypothèse sur la validité de la fonction T, à savoir que seuls `true` et `false` sont typés comme des booléens. Autrement dit, le schéma de type renvoyé par $T(\mathtt{c})$ doit être incompatible avec `bool` pour toutes les constantes autres que `true` et `false`.

  Les trois cas qui viennent, concernant l'évaluation d'un `match` fermé (c'est-à-dire sans cas par défaut), sont très similaires aux cas précédents concernant le `if` :
- Cas « $\mathtt{e}_b = \mathtt{match}\ \ \lambda\,\mathtt{x}\,.\,\mathtt{e}\ \mathtt{with}\ \ \mathtt{K}_1\ \mathtt{x}_1 \to \mathtt{e}_1 \parallel\ \ldots\ \parallel\ \mathtt{K}_n\ \mathtt{x}_n \to \mathtt{e}_n$ » :
  Si l'arbre d'inférence était constructible, il serait de la forme :

$$\textsc{TMatch}\ \dfrac{\dfrac{}{\varnothing \vdash \alpha_e \leqslant \{\ \mathtt{K}_1\ \alpha_1 \parallel\ \ldots\ \parallel\ \mathtt{K}_n\ \alpha_n\ \} \rhd \Phi'}\,T_1 \qquad \dfrac{\dfrac{}{\ldots} \quad \dfrac{}{\Phi'' \vdash \alpha'_1 \to \alpha'_2 \leqslant \alpha_e \rhd \Phi'''}\,T_2}{\Phi', \varnothing \vdash \lambda\mathtt{x.e} : \alpha_e \rhd \Phi'''} \quad \dfrac{}{\ldots} \qquad \dfrac{}{\ldots}}{\varnothing, \varnothing \vdash \mathtt{match}\ \lambda\mathtt{x.e}\ \mathtt{with}\ \mathtt{K}_1\ \mathtt{x}_1 \to \mathtt{e}_1 \parallel\ \ldots\ \parallel\ \mathtt{K}_n\ \mathtt{x}_n \to \mathtt{e}_n : \alpha \rhd \Phi'''}$$

  L'application de la transitivité de $(\leqslant)$ via la règle STransRight engendre la contrainte $(\alpha'_1 \to \alpha'_2 \leqslant \{\ \mathtt{K}_1\ \alpha_1 \parallel\ \ldots\ \parallel\ \mathtt{K}_n\ \alpha_n\ \})$ sur laquelle aucune règle d'inférence ne s'applique.

- Cas « $\mathtt{e}_b = \mathtt{match}\ (\mathtt{v}_1, \mathtt{v}_2)\ \mathtt{with}\ \ \mathtt{K}_1\ \mathtt{x}_1 \to \mathtt{e}_1 \parallel\ \ldots\ \parallel\ \mathtt{K}_n\ \mathtt{x}_n \to \mathtt{e}_n$ » :
  Ce cas est similaire au précédent. La tentative d'inférence de types pour ce programme engendre une contrainte de la forme $(\alpha'_1 \times \alpha'_2 \leqslant \{\ \mathtt{K}_1\ \alpha_1 \parallel\ \ldots\ \parallel\ \mathtt{K}_n\ \alpha_n\ \})$ sur laquelle aucune règle de saturation ne s'applique.

- Cas « $\mathtt{e}_b = \mathtt{match\ c\ with}\ \ \mathtt{K}_1\ \mathtt{x}_1 \to \mathtt{e}_1 \parallel\ \ldots\ \parallel\ \mathtt{K}_n\ \mathtt{x}_n \to \mathtt{e}_n$ » :
  Pour ce cas, nous avons encore une fois besoin d'une propriété de validité de T associée au fait qu'aucune constante n'est un constructeur de données : quelle que soit la constante `c`, la fonction T appliquée à `c` doit générer un schéma de type incompatible avec un ensemble clos de variants (c'est-à-dire de la forme $\{\ \mathtt{K}_1\ \alpha_1 \parallel\ \ldots\ \parallel\ \mathtt{K}_n\ \alpha_n\ \}$).

  Tous les cas restants concernent une expression qui est bloquée parce qu'une de ses sous-expressions est bloquée. Pour rappel, l'existence d'une sous-expression bloquée n'entraîne pas toujours le blocage de l'expression englobante. Le contexte d'évaluation `E` définit **la** prochaine sous-expression à évaluer et si l'expression englobante est bloquée, ce ne peut être qu'à cause d'elle.

- Cas « $\mathtt{e}_b = \mathtt{e}_1\ \mathtt{e}_2$ avec $\mathtt{e}_1$ bloquée » :
  La seule règle de typage applicable sur une telle expression est TApp, qui impose de typer $\mathtt{e}_1$. Or, $\mathtt{e}_1$ est bloquée, donc non-typable par hypothèse d'induction.

- Cas « $\mathtt{e}_b = \mathtt{v}\ \mathtt{e}$ avec $\mathtt{e}$ bloquée » :
  Similaire au cas précédent.

- Cas « $\mathtt{e}_b = \mathtt{p}^1\ \mathtt{e}$ avec $\mathtt{e}$ bloquée » :
  La seule règle de typage applicable sur une telle expression est TAppPrim1 qui impose de typer $\mathtt{e}$. L'expression $\mathtt{e}$ est bloquée donc non-typable par hypothèse d'induction.

- Cas « $\mathtt{e}_b = \mathtt{p}^2\ \mathtt{e}_1\ \mathtt{e}_2$ avec $\mathtt{e}_1$ bloquée » :
  Similaire au cas précédent avec la règle TAppPrim2.

- Cas « $\mathtt{e}_b = \mathtt{p}^2\ \mathtt{v}\ \mathtt{e}$ avec $\mathtt{e}$ bloquée » :
  Similaire au cas précédent.

- Cas « $\mathtt{e}_b = (\mathtt{e}_1, \mathtt{e}_2)$ avec $\mathtt{e}_1$ bloquée » :
  Similaire au cas précédent avec la règle TPair.

- Cas « $\mathtt{e}_b = (\mathtt{v}, \mathtt{e})$ avec $\mathtt{e}$ bloquée » :
  Similaire au cas précédent.

- Cas « $\mathtt{e}_b = \mathtt{K}\ \mathtt{e}$ avec $\mathtt{e}$ bloquée » :
  Similaire au cas précédent avec la règle TConstr.

- Cas « $\mathtt{e}_b = \mathtt{let}\ \mathtt{x} = \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2$ avec $\mathtt{e}_1$ bloquée » :
  Similaire au cas précédent avec la règle TLet.

- Cas « $\mathtt{e}_b = \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3$ avec $\mathtt{e}_1$ bloquée » :
  Similaire au cas précédent avec la règle TIf.

- Cas « $\mathtt{e}_b = \mathtt{match}\ \mathtt{e}\ \mathtt{with}\ \ldots$ avec $\mathtt{e}$ bloquée » :
  Similaire au cas précédent avec la règle TMatch ou TMatchDefault en fonction de la présence d'un cas par défaut ou non dans le filtrage.

Lemme 9 (Évaluation uniforme)**.**

Toute expression $\mathtt{e}$ *vérifie une et une seule des propriétés suivantes :*

- *L'évaluation de* $\mathtt{e}$ *boucle indéfiniment*
- *Il existe une valeur* $\mathtt{v}$ *telle que* $\mathtt{e} \longmapsto\!\!\!\!\!\rightarrow\ \mathtt{v}$
- *Il existe une expression bloquée* $\mathtt{e}_b$ *telle que* $\mathtt{e} \longmapsto\!\!\!\!\!\rightarrow\ \mathtt{e}_b$.

Démonstration (Lemme 9) :

Si nous ne sommes pas dans le premier cas du lemme, et que donc l'évaluation ne boucle pas, cela signifie qu'il existe une expression e' telle que $\mathsf{e} \longmapsto\!\!\!\!\!\rightarrow \mathsf{e}'$ et aucune expression e'' telle que $\mathsf{e}' \longmapsto \mathsf{e}''$. Nous distinguons alors deux cas :

- Soit e' est une valeur, nous sommes alors dans le second cas du lemme.
- Soit e' n'est pas une valeur, c'est donc une expression bloquée par définition, et nous sommes dans le troisième cas du lemme.

Théorème 2 (Validité forte)**.**

Pour toute expression e *et tout schéma de type* σ*, si* e *et* σ *vérifient* $\mathsf{e} : \sigma$ *alors l'une des propriétés suivantes est vérifiée :*

- *L'évaluation de* e *boucle indéfiniment.*
- *L'expression* e *s'évalue en la valeur* v *(c'est-à-dire* $\mathsf{e} \longmapsto\!\!\!\!\!\rightarrow \mathsf{v}$*) et* $\mathsf{v} : \sigma$*.*

Démonstration (Théorème 2) :

Nous démontrons tout d'abord que le dernier cas du lemme 9 (évaluation uniforme) est impossible lorsqu'il existe σ tel que $(\mathsf{e} : \sigma)$.
Supposons qu'il existe :

- e et σ vérifiant $(\mathsf{e} : \sigma)$
- α une variable de type
- e_b une expression bloquée telle que $\mathsf{e} \longmapsto\!\!\!\!\!\rightarrow \mathsf{e}_b$

Et déduisons-en une absurdité.
Par définition de la relation $(:)$, l'expression e est typable avec α dans $\varnothing, \varnothing$ en produisant un ensemble de contraintes Φ vérifiant $\textsc{gen}(\alpha, \Phi, \varnothing) \leqslant \sigma$:

$$\varnothing, \varnothing \vdash \mathsf{e} : \alpha \rhd \Phi$$

D'après le lemme 7, puisque $\mathsf{e} \longmapsto\!\!\!\!\!\rightarrow \mathsf{e}_b$, et e est typable, l'expression e_b est typable. Or, e_b est bloquée et donc non-typable d'après le lemme 8. Contradiction.

Montrons maintenant que lorsque $\mathsf{e} \longmapsto\!\!\!\!\!\rightarrow \mathsf{v}$ avec v une valeur, alors v vérifie la relation $(\mathsf{v} : \sigma)$.
D'après le lemme 7, la valeur v est typable dans $\varnothing, \varnothing$ en générant un ensemble de contraintes Φ' vérifiant $\Phi' \leqslant \Phi$:

$$\varnothing, \varnothing \vdash v : \alpha \rhd \Phi'$$

Comme $\Phi' \leqslant \Phi$, nous avons bien $\textsc{gen}(\alpha, \Phi', \varnothing) \leqslant \sigma$. Par définition de la relation $(:)$, nous pouvons conclure directement $(\mathsf{v} : \sigma)$

Théorème 1 (Validité faible).

Pour toute expression `e` *et tout schéma de type* σ*, si* `e` *et* σ *vérifient* `e` : σ *alors* `eval(e)` $\neq$ `ERROR`.

Démonstration (Théorème 1) :

Il s'agit d'un simple corollaire du théorème de validité forte. Par définition, `eval(e)` = `ERROR` lorsqu'il existe une expression bloquée $\mathtt{e}_b$ telle que `e` $\longmapsto\!\!\!\!\!\rightarrow$ $\mathtt{e}_b$. Puisqu'il existe σ tel que (`e` : σ), cette situation est impossible d'après le théorème 2.

2.5 Gestion de la mutabilité

Lorsque le langage que nous manipulons propose des structures de données mutables, que ce soit via des constructions de syntaxe (comme des enregistrements mutables par exemple) ou simplement via des types abstraits et leurs primitives associées (comme les tableaux dans certains langages comme OCaml), il est nécessaire de modifier légèrement le système de types pour gérer correctement le polymorphisme.

2.5.1 Modification du système de types

Le typage actuel du `let` est incorrect en présence de mutabilité (cf. [LW91]). Pour résoudre ce problème, nous utilisons la technique très classique dite de la « value restriction » (cf. [W95, G04[2]]). Essentiellement, elle consiste à n'effectuer une généralisation lors du typage d'une expression de la forme (`let x` = $\mathtt{e}_1$ `in` $\mathtt{e}_2$) uniquement lorsque $\mathtt{e}_1$ est « non-expansive », au sens où il n'y a pas de « calcul » à effectuer pour l'évaluer. Les expressions non-expansives sont, dans notre langage, les variables, les constantes, les λ et les `let` dans lesquels les deux sous-expressions sont non-expansives. À l'inverse, une application de fonction, une application de primitive, un filtrage de motifs et un `let` dont l'une des sous-expression est expansive sont des expressions « expansives ».

La règle de typage à appliquer sur une expression de la forme (`let x` = $\mathtt{e}_1$ `in` $\mathtt{e}_2$) où $\mathtt{e}_1$ est une expression expansive est beaucoup plus simple que la règle du `let` polymorphe :

TLetExpansive

$$\frac{\texttt{when } \mathtt{e}_1 \texttt{ is expansive} \qquad \texttt{let } \alpha' \textit{ fresh} \qquad \Phi_1, \Gamma \vdash \mathtt{e}_1 : \alpha' \rhd \Phi_2 \qquad \Phi_2, \Gamma \oplus (\mathtt{x} : \alpha') \vdash \mathtt{e}_2 : \alpha \rhd \Phi_3}{\Phi_1, \Gamma \vdash \texttt{let } \mathtt{x} = \mathtt{e}_1 \texttt{ in } \mathtt{e}_2 : \alpha \rhd \Phi_3}$$

2.5.2 De nouvelles primitives

Pour ajouter de la mutabilité dans notre langage, nous allons définir des « références ». Plutôt que d'introduire une nouvelle construction de langage, nous préférons simplement définir un type paramétré noté $(\alpha\ \texttt{ref})$, et trois primitives :

- `ref` : une primitive unaire prenant une valeur en argument et créant une référence contenant cette valeur. Le schéma de type qui lui est associé, que l'on « pretty-print » en général sous la forme $(\alpha \to \alpha\ \texttt{ref})$, est :

  $$T(\texttt{ref}) \triangleq [\ \forall \alpha\alpha_0\alpha_1\ .\ \alpha \mid \alpha_0 \to \alpha_1 \leqslant \alpha\ \wedge\ \alpha_0\ \texttt{ref} \leqslant \alpha_1\ ]$$

- `get` : une primitive unaire prenant une référence en argument et renvoyant la valeur contenue dans cette référence. Son schéma de type associé, en général pretty-printé sous la forme $(\alpha\ \texttt{ref} \to \alpha)$, est :

  $$T(\texttt{get}) \triangleq [\ \forall \alpha\alpha_0\alpha_1\ .\ \alpha \mid \alpha_0 \to \alpha_1 \leqslant \alpha\ \wedge\ \alpha_0 \leqslant \alpha_1\ \texttt{ref}\ ]$$

- `set` : une primitive binaire prenant une référence et une valeur en arguments, modifiant le contenu de la référence avec la valeur donnée, et renvoyant la valeur « () ». Son schéma de type associé, en général pretty-printé sous la forme $(\alpha\ \texttt{ref} \to \alpha \to \texttt{unit})$, est :

  $$T(\texttt{set}) \triangleq [\ \forall \alpha\alpha_0\alpha_1\alpha_2\alpha_3\ .\ \alpha \mid \alpha_0 \to \alpha_1 \leqslant \alpha \wedge \alpha_2 \to \alpha_3 \leqslant \alpha_1 \wedge \alpha_2 \leqslant \alpha_0\ \texttt{ref} \wedge \texttt{unit} \leqslant \alpha_3\ ]$$

2.5.3 Exemple

Observons alors le comportement de notre algorithme de typage sur le code suivant créant une référence sur des variants :

```
let rcolor = ref Jaune in
if rand () then set rcolor Bleu;
let color = get rcolor in
...
```

où :

- La fonction « `rand` » renvoie simplement un booléen de manière aléatoire.
- La construction « `if` $\texttt{e}_1$ `then` $\texttt{e}_2$ » est du sucre syntaxique sur « `if` $\texttt{e}_1$ `then` $\texttt{e}_2$ `else ()` ».
- La construction de séquence « $\texttt{e}_1$`;` $\texttt{e}_2$ » est du sucre syntaxique sur « `let x =` $\texttt{e}_1$ `in` $\texttt{e}_2$ » où « `x` » est une variable inutilisée ailleurs dans le programme.

Comme l'expression `(ref Jaune)` est expansive, le typage du premier `let` se fait via la règle TLetExpansive. Le typage de `(ref Jaune :` α`)` engendre alors (entre autres) les contraintes suivantes :

- $\alpha_0\ \texttt{ref} \leqslant \alpha$
- `Jaune` $\leqslant \alpha_0$

La variable `rcolor` est alors associée à la variable de type α dans l'environnement de typage utilisé pour typer le reste du code. Le typage de `(set rcolor Bleu :` α'`)` contraint donc également la variable α_0 en engendrant :

- `Bleu` $\leqslant \alpha_0$

La variable de type α'' associée à la variable `color` lors du typage du reste du code est alors contrainte lors du typage de `(get rcolor :` α''`)` qui engendre bien les deux contraintes :

- `Jaune` $\leqslant \alpha''$
- `Bleu` $\leqslant \alpha''$

spécifiant correctement que la variable `color` peut valoir soit le variant `Jaune`, soit le variant `Bleu`.

2.5.4 Remarques sur la variance

Contrairement aux systèmes classiques, nous n'utilisons dans nos systèmes aucune notion de « variance » pour gérer des contraintes liant des types paramétrés. L'utilisation de la règle de saturation STypeConstr est nécessaire pour gérer les comparaisons entre deux types paramétrés comme par exemple $(\alpha\ \texttt{ref} \leqslant \alpha'\ \texttt{ref})$. Les deux contraintes $(\alpha \leqslant \alpha')$ et $(\alpha' \leqslant \alpha)$ sont alors générées, ce qui peut être vu comme une forme d'« invariance » du type `ref`. Cette propagation de la relation de sous-typage dans les deux sens n'est toutefois pas spécifique au type `ref` mais est effectuée sur tous les types paramétrés, y compris $(\rightarrow)$, sans que cela restreigne l'expressivité de nos systèmes.

Dans nos systèmes, les « propriétés de variance » des paramètres des constructeurs de type n'est pas encodée comme une information supplémentaire sur ces constructeurs. Elle est simplement définie par les contraintes générées au moment du typage. En particulier, les « propriétés de variance » des paramètres de la flèche $(\rightarrow)$ proviennent de :

- La règle :

$$\textsc{TLambda}\quad \frac{\texttt{let}\ \alpha_1, \alpha_2\ \textit{fresh} \qquad \Phi, \Gamma \oplus (\mathtt{x}, \alpha_1) \vdash \mathtt{e} : \alpha_2 \rhd \Phi' \qquad \Phi' \vdash \alpha_1 \to \alpha_2 \leqslant \alpha \rhd \Phi''}{\Phi, \Gamma \vdash \lambda\ \mathtt{x}\ .\ \mathtt{e} : \alpha \rhd \Phi''}$$

  Cette règle génère une flèche $(\alpha_1 \to \alpha_2)$ à gauche d'une relation de sous-typage qui restera toujours à gauche pendant la saturation. La variable de type α_1 est alors associée à la variable `x` dans l'environnement et est donc uniquement contrainte par la droite lors du typage des différentes occurrences de `x` dans `e`. La variable α_2 est quant à elle utilisée pour le typage de $(\mathtt{e} : \alpha_2)$ et est donc uniquement contrainte par la gauche.

- La règle :

$$\textsc{TApp}\quad \frac{\begin{array}{c}\texttt{let}\ \alpha_1, \alpha_2, \alpha_3\ \textit{fresh} \qquad \Phi \vdash \alpha_1 \leqslant \alpha_2 \to \alpha_3 \rhd \Phi' \qquad \Phi' \vdash \alpha_3 \leqslant \alpha \rhd \Phi'' \\ \Phi'', \Gamma \vdash \mathtt{e}_1 : \alpha_1 \rhd \Phi''' \qquad \Phi''', \Gamma \vdash \mathtt{e}_2 : \alpha_2 \rhd \Phi''''\end{array}}{\Phi, \Gamma \vdash \mathtt{e}_1\ \mathtt{e}_2 : \alpha \rhd \Phi''''}$$

Cette règle génère une flèche $(\alpha_2 \to \alpha_3)$ à droite d'une relation de sous-typage qui restera toujours à droite pendant la saturation. La variable α_2 est alors uniquement contrainte par la gauche lors du typage de $(\mathtt{e}_2 : \alpha_2)$ et la variable α_3 est quant à elle contrainte par la droite à cause de la contrainte $(\alpha_3 \leqslant \alpha)$.

- Les règles TAppPrim1 et TAppPrim2 qui sont similaires à TApp.

Par ailleurs, les « propriétés de variance » du paramètre du type `ref` proviennent des propriétés des schémas de type associés aux primitives `ref`, `get` et `set` :

- Dans le cas des primitives `ref` et `set` qui « écrivent » dans une référence, le paramètre α_0 du type $(\alpha_0\ \mathtt{ref})$ apparaît dans $T(\mathtt{ref})$ et $T(\mathtt{set}\ )$ uniquement à gauche d'une flèche située à gauche d'un $(\leqslant)$. D'après les propriétés que nous avons données sur les flèches, cette variable α_0 ne pourra alors être contrainte que par la gauche lors de la saturation.
- Dans le cas de GET qui « lit » dans une référence, le paramètre α_1 du type $(\alpha_1\ \mathtt{ref})$ apparaît dans $T(\mathtt{get}\ )$ uniquement à droite d'une flèche située à gauche d'un $(\leqslant)$. La variable α_1 ne pourra alors être contrainte que par la droite lors de la saturation.

En conclusion, la « variance » dans nos systèmes est intrinsèque aux contraintes générées sur les paramètres des types et ne nécessite pas de traitement particulier lors de la saturation.

2.6 Renforcement du typage

2.6.1 Motivation

Le système de types de base que nous avons présenté dans ce chapitre est valide mais présente un défaut d'ordre conceptuel pouvant poser quelques problèmes dans son usage pratique dans le cadre d'une implémentation concrète. En effet, la technique de saturation que nous avons définie autorise certaines contraintes, comme typiquement $(\alpha \leqslant \mathtt{int})$ et $(\alpha \leqslant \mathtt{string})$, à coexister dans le même Φ sans provoquer de clash. Cependant, le α en question ne peut alors plus être « contraint par la gauche » : il est impossible d'ajouter à Φ une nouvelle contrainte de la forme $\tau^l \leqslant \alpha$, avec τ^l un type paramétré, sans provoquer un clash pendant la saturation.

Une telle propriété du mécanisme de saturation, sans pour autant être « invalide » du point de vue de la sémantique du langage, autorise le programmeur à définir très facilement des fonctions ne pouvant jamais être appelées, comme par exemple le `f` du code suivant :

```
let f x = (x + 1, x ^ " world!") in
...
```

pour lequel le schéma de type inféré est de la forme :

$$\mathtt{f} : [\ \forall\alpha_0\alpha_1\alpha_2\ .\ \alpha_0 \mid \alpha_1 \to \alpha_2 \leqslant \alpha_0\ \wedge\ \alpha_1 \leqslant \mathtt{int}\ \wedge\ \alpha_1 \leqslant \mathtt{string}\ \wedge\ \ldots\ ]$$

Il n'est pas très « pratique » pour le programmeur d'autoriser le typage d'un tel début de code et de retarder le clash, dont l'origine est l'utilisation du paramètre `x` dans des contextes incompatibles dans le corps de `f`, à la première tentative d'appel de `f`.

Pour éviter ce genre de désagrément, il est possible de « renforcer » le système de types dans le but d'obtenir un système moins puissant (au sens où il accepte moins de programmes valides), mais dont l'implémentation en un typeur génère des messages d'erreur « souvent mieux localisés ».

2.6.2 Une nouvelle relation : $(\approx)$

Nous définissons alors une nouvelle relation symétrique, que nous notons $(\approx)$, représentant la « compatibilité » entre deux types. Pour ce faire, nous étendons la grammaire des contraintes de la manière suivante :

$$\texttt{C} \texttt{ +:= } \tau_1^l \approx \tau_2^l \mid \tau_1^r \approx \tau_2^r$$

Lors de la saturation, en plus des autres opérations liées à la transitivité de la relation $(\leqslant)$, chaque fois que l'on rencontre deux contraintes de la forme $(\alpha \leqslant \tau_1^r)$ et $(\alpha \leqslant \tau_2^r)$, on engendre la contrainte $(\tau_1^r \approx \tau_2^r)$. De manière symétrique, chaque fois que l'on rencontre deux contraintes de la forme $(\tau_1^l \leqslant \alpha)$ et $(\tau_2^l \leqslant \alpha)$, on engendre la contrainte $(\tau_1^l \approx \tau_2^l)$.

2.6.3 Adaptation de la saturation

Nous commençons par définir deux nouvelles fonctions, notées « LCOMPATS » et « RCOMPATS », extrayant d'un ensemble de contraintes tous les types liés par une relation de compatibilité ou de sous-typage à une variable de type donnée :

$$\begin{aligned}
\textsc{lcompats}(\alpha, \Phi) &\triangleq \{\ \tau^l \mid (\tau^l \approx \alpha) \in \Phi \ \vee\ (\alpha \approx \tau^l) \in \Phi \ \vee\ (\tau^l \leqslant \alpha) \in \Phi\ \} \\
\textsc{rcompats}(\alpha, \Phi) &\triangleq \{\ \tau^r \mid (\tau^r \approx \alpha) \in \Phi \ \vee\ (\alpha \approx \tau^r) \in \Phi \ \vee\ (\alpha \leqslant \tau^r) \in \Phi\ \}
\end{aligned}$$

Il suffit alors de modifier les trois règles suivantes gérant l'arrivée d'une contrainte de sous-typage faisant intervenir une variable de type :

STRANSRIGHT

$$\frac{\begin{array}{c}\texttt{when } \tau^l \notin \{\ \alpha\ \} \qquad \texttt{let } \tau_1^r, \ldots, \tau_n^r = \textsc{rights}(\alpha, \Phi_1) \qquad \texttt{let } \tau_1^l, \ldots, \tau_p^l = \textsc{lcompats}(\alpha, \Phi_1) \\ \Phi_1 \vdash \tau^l \leqslant \tau_1^r \rhd \Phi_2 \qquad \cdots \qquad \Phi_n \vdash \tau^l \leqslant \tau_n^r \rhd \Phi_{n+1} \\ \Phi_{n+1} \vdash \tau^l \approx \tau_1^l \rhd \Phi_{n+2} \qquad \cdots \qquad \Phi_{n+p} \vdash \tau^l \approx \tau_p^l \rhd \Phi_{n+p+1}\end{array}}{\Phi_1 \vdash \tau^l \leq \alpha \rhd \Phi_{n+p+1}}$$

STRANSLEFT

$$\frac{\begin{array}{c}\texttt{when } \tau^r \notin \{\ \alpha\ \} \qquad \texttt{let } \tau_1^l, \ldots, \tau_n^l = \textsc{lefts}(\alpha, \Phi_1) \qquad \texttt{let } \tau_1^r, \ldots, \tau_p^r = \textsc{rcompats}(\alpha, \Phi_1) \\ \Phi_1 \vdash \tau_1^l \leqslant \tau^r \rhd \Phi_2 \qquad \cdots \qquad \Phi_n \vdash \tau_n^l \leqslant \tau^r \rhd \Phi_{n+1} \\ \Phi_{n+1} \vdash \tau_1^r \approx \tau^r \rhd \Phi_{n+2} \qquad \cdots \qquad \Phi_{n+p} \vdash \tau_p^r \approx \tau^r \rhd \Phi_{n+p+1}\end{array}}{\Phi_1 \vdash \alpha \leq \tau^r \rhd \Phi_{n+p+1}}$$

$$\frac{\begin{array}{l}\textsc{STransLeftRight}\\ \texttt{when}\ \alpha_1 \neq \alpha_2 \qquad \texttt{let}\ \tau_1^l, \ldots, \tau_n^l = \textsc{lefts}(\alpha_1,\ \Phi_{1,1}) \qquad \texttt{let}\ \tau_1^r, \ldots, \tau_p^r = \textsc{rights}(\alpha_2,\ \Phi_{1,1})\end{array} \quad \begin{array}{ccc}\Phi_{1,1} \vdash \tau_1^l \leqslant \tau_1^r \rhd \Phi_{1,2} & \cdots & \Phi_{1,p} \vdash \tau_1^l \leqslant \tau_p^r \rhd \Phi_{2,1} \\ \vdots & & \vdots \\ \Phi_{n,1} \vdash \tau_n^l \leqslant \tau_1^r \rhd \Phi_{n,2} & \cdots & \Phi_{n,p} \vdash \tau_n^l \leqslant \tau_p^r \rhd \Phi_{n+1,1}\end{array}}{\Phi_{n+1,1} \vdash \alpha_1 \approx \alpha_2 \rhd \Phi'}$$
$$\Phi_{1,1} \vdash \alpha_1 \leq \alpha_2 \rhd \Phi'$$

Il faut maintenant enrichir le système avec de nouvelles règles gérant ces nouvelles contraintes de compatibilité. Pour interdire les arbres infinis, et donc éviter de boucler lors de la saturation des contraintes, nous introduisons un mécanisme de calcul de point fixe similaire à celui gérant la génération des contraintes de sous-typage. Ce mécanisme est implémenté par les deux règles suivantes :

$$\frac{\begin{array}{l}\textsc{SNewCompat}\\ \texttt{when}\ (\tau_1 \approx \tau_2) \notin \Phi \qquad \Phi \wedge \tau_1 \approx \tau_2 \vdash \tau_1 \simeq \tau_2 \rhd \Phi'\end{array}}{\Phi \vdash \tau_1 \approx \tau_2 \rhd \Phi'}$$

gérant l'arrivée d'une nouvelle contrainte de compatibilité inconnue jusqu'alors, et :

$$\frac{\begin{array}{l}\textsc{SCompatAlreadyProved}\\ \texttt{when}\ (\tau_1 \approx \tau_2) \in \Phi\end{array}}{\Phi \vdash \tau_1 \approx \tau_2 \rhd \Phi}$$

gérant l'arrivée d'une contrainte de compatibilité déjà connue.

Ces deux règles sont valables pour des τ^l comme pour des τ^r. Nous remarquerons néanmoins, comme nous l'impose la grammaire des contraintes (C), que les deux membres des contraintes de compatibilité engendrées par notre système sont toujours de la même nature : il s'agit soit de deux τ^l, soit de deux τ^r.

De manière similaire à la transformation de la relation $(\leqslant)$ en $(\leq)$, lorsqu'une nouvelle contrainte de la forme $(\tau_1 \approx \tau_2)$ arrive, la contrainte $(\tau_1 \simeq \tau_2)$ est générée et est gérée par les règles que nous allons définir maintenant.

Une contrainte de compatibilité entre deux types paramétrés n'est autorisée que lorsque ces deux types sont de même nom, et est propagée sur leurs paramètres. Ce comportement est implémenté par la règle suivante :

$$\frac{\begin{array}{l}\textsc{SCompatParamed}\\ \Phi_1 \vdash \alpha_1 \approx \alpha_1' \rhd \Phi_2 \qquad \cdots \qquad \Phi_n \vdash \alpha_n \approx \alpha_n' \rhd \Phi_{n+1}\end{array}}{\Phi_1 \vdash (\alpha_1, \ldots, \alpha_n)\ \mathtt{t} \simeq (\alpha_1', \ldots, \alpha_n')\ \mathtt{t} \rhd \Phi_{n+1}}$$

Une contrainte de compatibilité entre deux constructeurs de données de même nom est propagée à leur paramètre :

$$\textsc{SCompatSameConstr}\ \frac{\Phi \vdash \alpha \approx \alpha' \rhd \Phi'}{\Phi \vdash \mathtt{K}\ \alpha \simeq \mathtt{K}\ \alpha' \rhd \Phi'}$$

Une contrainte de compatibilité entre deux constructeurs de données de noms différents est quant à elle toujours valide :

$$\textsc{SCompatDiffConstr}\ \frac{\texttt{when}\ \mathtt{K}_1 \neq \mathtt{K}_2}{\Phi \vdash \mathtt{K}_1\ \alpha \simeq \mathtt{K}_2\ \alpha' \rhd \Phi}$$

Pour que deux ensembles de constructeurs soient compatibles, il faut que leurs éléments le soient deux à deux :

$$\textsc{SCompatConstrSet}\ \frac{\begin{array}{ccc}\Phi_{1,1} \vdash \mathtt{K}_1\ \alpha_1 \approx \mathtt{K}'_1\ \alpha'_1 \rhd \Phi_{1,2} & \cdots & \Phi_{1,p} \vdash \mathtt{K}_1\ \alpha_1 \approx \mathtt{K}'_p\ \alpha'_p \rhd \Phi_{2,1} \\ \vdots & & \vdots \\ \Phi_{n,1} \vdash \mathtt{K}_n\ \alpha_n \approx \mathtt{K}'_1\ \alpha'_1 \rhd \Phi_{n,2} & \cdots & \Phi_{n,p} \vdash \mathtt{K}_n\ \alpha_n \approx \mathtt{K}'_p\ \alpha'_p \rhd \Phi_{n+1,1}\end{array}}{\Phi_{1,1} \vdash \{\ \mathtt{K}_1\ \alpha_1 \parallel \ \ldots\ \parallel \mathtt{K}_n\ \alpha_n\ \} \simeq \{\ \mathtt{K}'_1\ \alpha'_1 \parallel \ \ldots\ \parallel \mathtt{K}'_p\ \alpha'_p\ \} \rhd \Phi_{n+1,1}}$$

Bien entendu, une implémentation optimisée de cette règle effectuera un simple parcours linéaire des deux ensembles de constructeurs triés par nom.

Les trois règles suivantes sont très similaires. Elles gèrent la présence éventuelle d'un cas par défaut dans les ensembles de contraintes et propage la compatibilité sur la variable de type associée si elle est présente des deux côtés :

$$\textsc{SCompatConstrSetDN}\ \frac{\begin{array}{ccc}\Phi_{1,1} \vdash \mathtt{K}_1\ \alpha_1 \approx \mathtt{K}'_1\ \alpha'_1 \rhd \Phi_{1,2} & \cdots & \Phi_{1,p} \vdash \mathtt{K}_1\ \alpha_1 \approx \mathtt{K}'_p\ \alpha'_p \rhd \Phi_{2,1} \\ \vdots & & \vdots \\ \Phi_{n,1} \vdash \mathtt{K}_n\ \alpha_n \approx \mathtt{K}'_1\ \alpha'_1 \rhd \Phi_{n,2} & \cdots & \Phi_{n,p} \vdash \mathtt{K}_n\ \alpha_n \approx \mathtt{K}'_p\ \alpha'_p \rhd \Phi_{n+1,1}\end{array}}{\Phi_{1,1} \vdash \{\ \mathtt{K}_1\ \alpha_1 \parallel \ \ldots\ \parallel \mathtt{K}_n\ \alpha_n \parallel \alpha\ \} \simeq \{\ \mathtt{K}'_1\ \alpha'_1 \parallel \ \ldots\ \parallel \mathtt{K}'_p\ \alpha'_p\ \} \rhd \Phi_{n+1,1}}$$

$$\textsc{SCompatConstrSetND}\ \frac{\begin{array}{ccc}\Phi_{1,1} \vdash \mathtt{K}_1\ \alpha_1 \approx \mathtt{K}'_1\ \alpha'_1 \rhd \Phi_{1,2} & \cdots & \Phi_{1,p} \vdash \mathtt{K}_1\ \alpha_1 \approx \mathtt{K}'_p\ \alpha'_p \rhd \Phi_{2,1} \\ \vdots & & \vdots \\ \Phi_{n,1} \vdash \mathtt{K}_n\ \alpha_n \approx \mathtt{K}'_1\ \alpha'_1 \rhd \Phi_{n,2} & \cdots & \Phi_{n,p} \vdash \mathtt{K}_n\ \alpha_n \approx \mathtt{K}'_p\ \alpha'_p \rhd \Phi_{n+1,1}\end{array}}{\Phi_{1,1} \vdash \{\ \mathtt{K}_1\ \alpha_1 \parallel \ \ldots\ \parallel \mathtt{K}_n\ \alpha_n\ \} \simeq \{\ \mathtt{K}'_1\ \alpha'_1 \parallel \ \ldots\ \parallel \mathtt{K}'_p\ \alpha'_p \parallel \alpha'\ \} \rhd \Phi_{n+1,1}}$$

$$\textsc{SCompatConstrSetDD}\quad \frac{\begin{array}{ccc}\Phi_{1,1} \vdash \mathtt{K}_1\ \alpha_1 \approx \mathtt{K}'_1\ \alpha'_1 \rhd \Phi_{1,2} & \cdots & \Phi_{1,p} \vdash \mathtt{K}_1\ \alpha_1 \approx \mathtt{K}'_p\ \alpha'_p \rhd \Phi_{2,1} \\ \vdots & & \vdots \\ \Phi_{n,1} \vdash \mathtt{K}_n\ \alpha_n \approx \mathtt{K}'_1\ \alpha'_1 \rhd \Phi_{n,2} & \cdots & \Phi_{n,p} \vdash \mathtt{K}_n\ \alpha_n \approx \mathtt{K}'_p\ \alpha'_p \rhd \Phi_{n+1,1} \\ \multicolumn{3}{c}{\Phi_{n+1,1} \vdash \alpha \approx \alpha' \rhd \Phi'}\end{array}}{\Phi_{1,1} \vdash \{\ \mathtt{K}_1\ \alpha_1\ \|\ \ldots\ \|\ \mathtt{K}_n\ \alpha_n\ \|\ \alpha\ \} \simeq \{\ \mathtt{K}'_1\ \alpha'_1\ \|\ \ldots\ \|\ \mathtt{K}'_p\ \alpha'_p\ \|\ \alpha'\ \} \rhd \Phi'}$$

Enfin, il ne reste plus qu'à gérer la transitivité de la relation $(\approx)$, et donc à la propager à travers les variables de type. L'ajout d'une contrainte de compatibilité faisant intervenir une variable de type se fait alors grâce à cinq règles de saturation :

- Lorsque la contrainte de compatibilité relie une variable de type α à gauche et un τ^l qui n'est pas un τ^r à droite, elle est gérée par la règle suivante :

$$\textsc{SVarCompatLeft}\quad \frac{\begin{array}{cc}\mathtt{when}\ \tau^l \notin \{\ \tau^r\ \} & \mathtt{let}\ \tau^l_1, \ldots, \tau^l_n = \textsc{lcompats}(\alpha,\ \Phi_1) \\ \Phi_1 \vdash \tau^l_1 \approx \tau^l \rhd \Phi_2 & \cdots \qquad \Phi_n \vdash \tau^l_n \approx \tau^l \rhd \Phi_{n+1}\end{array}}{\Phi_1 \vdash \alpha \simeq \tau^l \rhd \Phi_{n+1}}$$

  On extrait alors tous les τ^l_i associés à α dans Φ, que ce soit par une relation de sous-typage comme par une relation de compatibilité, et on les contraint à être compatibles avec τ^l.

- De manière symétrique, lorsque le type est un τ^r qui n'est pas un τ^l, on extrait tous les τ^r_i liés à α et on impose leur compatibilité avec τ^r :

$$\textsc{SVarCompatRight}\quad \frac{\begin{array}{cc}\mathtt{when}\ \tau^r \notin \{\ \tau^l\ \} & \mathtt{let}\ \tau^r_1, \ldots, \tau^r_n = \textsc{rcompats}(\alpha,\ \Phi_1) \\ \Phi_1 \vdash \tau^r_1 \approx \tau^r \rhd \Phi_2 & \cdots \qquad \Phi_n \vdash \tau^r_n \approx \tau^r \rhd \Phi_{n+1}\end{array}}{\Phi_1 \vdash \alpha \simeq \tau^r \rhd \Phi_{n+1}}$$

- Lorsque le type est à la fois un τ^l et un τ^r mais pas une variable de type, on impose sa compatibilité avec tous les τ^l_i et tous les τ^r_i liés à la variable de type :

$$\textsc{SVarCompatLeftRight}\quad \frac{\begin{array}{c}\mathtt{when}\ \tau \in \{\ \tau^l\ \} \wedge \tau \in \{\ \tau^r\ \} \wedge \tau \notin \{\ \alpha\ \} \\ \mathtt{let}\ \tau_1, \ldots, \tau_n = \textsc{lcompats}(\alpha,\ \Phi_1) \cup \textsc{rcompats}(\alpha,\ \Phi_1) \\ \Phi_1 \vdash \tau_1 \approx \tau \rhd \Phi_2 \qquad \cdots \qquad \Phi_n \vdash \tau_n \approx \tau \rhd \Phi_{n+1}\end{array}}{\Phi_1 \vdash \alpha \simeq \tau \rhd \Phi_{n+1}}$$

- Lorsque la variable de type est à droite, on renverse la contrainte :

$$\textsc{SNonVarCompatVar}\quad \frac{\mathtt{when}\ \tau \notin \{\ \alpha\ \} \qquad \Phi \vdash \alpha \approx \tau \rhd \Phi'}{\Phi \vdash \tau \simeq \alpha \rhd \Phi'}$$

- Lorsque la relation de compatibilité relie deux variables de type, il faut aller chercher tous les types compatibles avec chacune d'entre elles et les lier par une relation de compatibilité :

$$\textsc{SVarCompatVar}$$
$$\texttt{when } \tau \notin \{\, \alpha \,\}$$
$$\texttt{let } \tau_1^l, \ldots, \tau_n^l = \textsc{lcompats}(\alpha_1,\ \Phi_{1,1}) \cup \textsc{lcompats}(\alpha_2,\ \Phi_{1,1})$$
$$\texttt{let } \tau_1^r, \ldots, \tau_p^r = \textsc{rcompats}(\alpha,\ \Phi_{1,1}) \cup \textsc{rcompats}(\alpha_2,\ \Phi_{1,1})$$

$$\begin{array}{ccc} \Phi_{1,1} \vdash \tau_1^l \approx \tau_1^l \rhd \Phi_{1,2} & \cdots & \Phi_{1,n} \vdash \tau_1^l \approx \tau_n^l \rhd \Phi_{2,1} \\ \vdots & & \vdots \\ \Phi_{n,1} \vdash \tau_n^l \approx \tau_1^l \rhd \Phi_{n,2} & \cdots & \Phi_{n,n} \vdash \tau_n^l \approx \tau_n^l \rhd \Phi'_{1,1} \end{array}$$

$$\begin{array}{ccc} \Phi'_{1,1} \vdash \tau_1^r \approx \tau_1^r \rhd \Phi'_{1,2} & \cdots & \Phi'_{1,p} \vdash \tau_1^r \approx \tau_p^r \rhd \Phi'_{2,1} \\ \vdots & & \vdots \\ \Phi'_{p,1} \vdash \tau_p^r \approx \tau_1^r \rhd \Phi'_{p,2} & \cdots & \Phi'_{p,p} \vdash \tau_p^r \approx \tau_p^r \rhd \Phi'' \end{array}$$
$$\Phi_{1,1} \vdash \alpha_1 \simeq \alpha_2 \rhd \Phi''$$

Cette dernière règle a quelques prémisses inutiles qui ont été écrites uniquement pour simplifier la numérotation. Lors de la saturation, il n'est en effet pas utile de générer des contraintes de compatibilité entre deux fois le même type comme $(\tau_1^r \approx \tau_1^r)$ (la relation de compatibilité étant naturellement réflexive), ni des contraintes de compatibilité dans les deux sens comme $(\tau_1^r \approx \tau_2^r)$ et $(\tau_2^r \approx \tau_1^r)$ (la relation de compatibilité étant symétrique). La génération de ces contraintes n'est néanmoins ni invalide, ni restrictive. Elle provoquerait uniquement une légère perte de performance si ce système était implémenté de manière naïve.

2.6.4 Remarques

Le mécanisme de renforcement du typage que nous avons décrit dans cette section peut être ajouté à tous les systèmes que nous présentons dans cette thèse. Il fait légèrement baisser les performances du typeur mais permet d'obtenir des systèmes plus « pratiques » à utiliser. Son interaction avec les disjonctions que nous allons introduire dans les chapitres 3 et 5 doit néanmoins être gérée correctement pour ne pas trop affaiblir le typage.

Nous remarquerons cependant que ce mécanisme casse une propriété intéressante du système de base. En effet, sans ce renforcement, il est possible d'associer tout clash de typage à un chemin d'exécution invalide puisqu'un clash provient systématiquement de la rencontre d'un τ^l (généré lors du typage de la création d'une valeur) avec un τ^r incompatible (généré lors du typage de la déconstruction d'une valeur). En revanche, comme nous l'avons vu, le renforcement du système décrit ici introduit de nouveaux clashs qui ne sont pas associés à des chemins d'exécution. Ils

peuvent en particulier provenir de deux utilisations d'un même paramètre de fonction dans deux contextes incompatibles, et ainsi relier deux points de déconstruction de valeurs. De tels clashs sont en général plus compliqués à expliquer au programmeur puisqu'ils ne sont pas reliés directement à l'exécution de leur code. Il s'agit plutôt d'une heuristique prévoyant qu'un morceau de programme, dont les limites sont difficiles à inférer automatiquement, est probablement mort.

Par ailleurs, il est parfaitement possible d'affaiblir ce renforcement du typage pour s'adapter à un environnement d'exécution plus « dynamique » permettant de « confondre » certains types de base, comme par exemple « `int` et `float` », ou bien « `bytes` et `string` » comme en OCaml (cf. [LD14]). Pour ce faire, il suffit d'ajouter des règles de typage spécifiques autorisant la compatibilité et/ou le sous-typage entre les types souhaités.

Nous ne faisons plus mention de ce mécanisme de renforcement dans les prochains chapitres, ni de la relation de compatibilité $(\approx)$ qui lui est associée. En effet, l'objectif à partir de maintenant est principalement d'étendre le système de types de base pour accepter des programmes actuellement refusés mais s'exécutant correctement.

Chapitre 3

Typage affiné du filtrage de motifs

Maintenant que nous avons posé le langage, sa sémantique, et un système de types de base pour ce langage, nous cherchons à étendre les mécanismes de typage dans le but d'accepter plus de programmes valides sans en accepter de faux. Les trois chapitres à venir abordent ce problème sous différents angles.

Dans celui-ci, nous cherchons à affiner le typage du filtrage de motifs sur les variants polymorphes. Pour ce faire, nous allons étendre le langage des contraintes de types, ainsi que la forme des règles de typage et de saturation. Malgré la « profondeur » de ces modifications, tout comme dans les prochains chapitres, ces transformations pourront être vues comme une « extension » du système de types de base car il existera chaque fois une « manière systématique » de transformer la plupart des règles de typage et de saturation du système de base pour aboutir au nouveau système considéré.

3.1 Contexte

Notre but ici est d'affiner le typage du filtrage de motifs en générant des contraintes indépendantes dans les différents cas de filtrage et en associant ces contraintes au cas considéré. Il s'agit en réalité d'une généralisation des travaux initiés par Aiken, Wimmers et Lakshman en 1994 (cf. [AW94]) puis repris par Pottier en 2000 (cf. [P00]) qui le présenta comme une instance de HM(X) (cf. [SP02]) puis par Garrigue en 2002 (cf. [G02]).

Ces différents travaux avaient pour objectif de permettre à un filtrage de motif de renvoyer des données de type différents dans les différents cas, et de lier le motif à ce type de retour.

Une telle opération peut alors être vue comme du « passage de message dynamique », l'idée sous-jacente étant d'encoder des « enregistrements » ou « objets » sans ajouter de nouvelle construction au langage comme le fait par exemple Rémy avec les enregistrements extensibles (cf. [R95]), mais en encodant simplement le dispatch dynamique grâce au filtrage.

3.2 Motivation

Comme nous l'avons vu, le système de types de base accepte déjà un code effectuant un filtrage et renvoyant des valeurs « de types différents » dans les différentes branches du filtrage comme

par exemple la fonction suivante :

```
let f = λ x . match x with
  ‖ I → 42
  ‖ S → "quarante deux" in
f I + 1
```

Le schéma de type inféré par le système de base pour une telle expression est :

$$\texttt{f} : [\, \forall \alpha\alpha_1\alpha_2 \,.\, \alpha \mid \alpha_1 \to \alpha_2 \leqslant \alpha \;\wedge\; \alpha_1 \leqslant \{\, \texttt{I} \parallel \texttt{S} \,\} \;\wedge\; \texttt{int} \leqslant \alpha_2 \;\wedge\; \texttt{string} \leqslant \alpha_2 \,]$$

Bien que correct, ce schéma de type est trop restrictif car il ne met pas en évidence le lien entre le paramètre de la fonction et la valeur retournée. Il est alors impossible de savoir que lorsque l'on passe le variant `I` à `f`, la valeur retournée sera un entier : en effet, si nous cherchons à typer `f I`, nous obtenons le schéma de type suivant :

$$\texttt{f I} : [\, \forall \alpha \,.\, \alpha \mid \texttt{int} \leqslant \alpha \;\wedge\; \texttt{string} \leqslant \alpha \,]$$

Une valeur annotée avec un tel schéma de type est presque « inutilisable » en pratique. En effet, il est impossible de la donner en argument d'un opérateur attendant un entier (la contrainte $(\texttt{string} \leqslant \alpha)$ provoquerait un clash) ni en argument d'une primitive attendant une chaîne de caractères (à cause de la contrainte $(\texttt{int} \leqslant \alpha)$). Pour « utiliser » une telle valeur, il faut soit la manipuler de manière complètement abstraite, soit disposer d'un moyen d'en extraire dynamiquement une information de typage.

Une telle approche a été implémentée dans le cadre de cette thèse et fonctionne comme on l'attend. L'objet de ce chapitre est néanmoins tout autre et beaucoup plus satisfaisant du point de vue de l'« analyse statique » comme nous allons le voir.

Pour résoudre ce problème statiquement, une approche standard consiste à « traquer » ces fonctions et à les refuser au moment du typage (ou a minima à émettre un « warning »). À l'inverse, nous allons ici chercher à accepter ce genre de fonction mais en lui inférant un schéma de type « plus précis ». En particulier, ce schéma nous permettra de déduire que, lorsque le variant `I` est passé en argument de `f`, la valeur renvoyée est un entier.

Dans un langage muni d'un système de types moderne comme OCaml ou Haskell, une technique classique pour écrire ce genre de code (c'est-à-dire une fonction renvoyant des valeurs de types différents selon la valeur de son argument) est d'utiliser un « **GADT** ». Bien qu'extrêmement puissants, les **GADT** apportent une certaine « lourdeur », à la fois dans l'écriture du code à cause des déclarations et annotations de types qu'ils nécessitent, mais également dans les messages d'erreur qu'il est difficile de rendre « clairs et intuitifs » pour le programmeur.

L'approche que nous proposons dans ce chapitre offre des fonctionnalités proches de celles fournies par les **GADT**, moins puissantes à cause de l'absence de « types existentiels », mais permettant

néanmoins de typer de nombreux codes intéressants en préservant l'inférabilité et une certaine simplicité. Nous verrons dans le chapitre 5 comment ajouter une variante des **GADT** à notre système pour les programmes où les types existentiels sont réellement utiles.

Une approche intuitive de ce problème consiste à étendre le langage des types avec de nouvelles constructions permettant de représenter le type de la fonction `f` de la manière suivante :

$$\texttt{f} : [\, \texttt{I} \to \texttt{int} \parallel \texttt{S} \to \texttt{string} \,]$$

La manipulation de telles constructions de types est malheureusement très compliquée car elles ont un statut particulier vis-à-vis du type classique des fonctions ($\to$) avec lequel elles sont partiellement compatibles. En particulier, un système de types voulant les faire cohabiter doit décider quand convertir un type fonctionnel en une telle construction de type, ou comment les comparer en perdant un minimum d'informations.

Plutôt que d'étendre le langage des types, nous choisissons ici d'étendre le langage des contraintes. Bien que moins intuitive, cette modification du langage des contraintes se montre en pratique beaucoup plus puissante et simple à mettre en oeuvre. Nous verrons par la suite que ce type de choix est pertinent dans d'autres contextes. C'est d'ailleurs cette approche qui a inspiré les extensions des systèmes de types présentés dans les prochain chapitres, cherchant toutes à étendre en premier lieu le langage des contraintes plutôt que le langage des types.

3.3 Modification du langage des contraintes

Nous ajoutons une nouvelle « relation », notée « # », entre un type τ^l et un constructeur de données :

$\texttt{C} \mathrel{+}:= \tau^l \ \texttt{\#} \ \texttt{K}$

Intuitivement, une telle relation $(\tau^l \ \# \ K)$ signifie qu'aucune valeur de la forme $(\texttt{K}\ \texttt{v})$ n'appartient à l'ensemble représenté par τ^l. Toujours intuitivement, la « loi » reliant les relations (#) et ($\leqslant$) est :

$$\tau^l \leqslant \alpha \ \wedge \ \alpha \,\#\, \texttt{K} \quad \Rightarrow \quad \tau^l \,\#\, \texttt{K}$$

ce qui est tout à fait cohérent du point de vue de la théorie des ensembles : si l'ensemble représenté par τ^l est inclus dans l'ensemble représenté par α et si α ne contient pas de valeur de la forme $(\texttt{K}\ \texttt{v})$, alors l'ensemble représenté par τ^l ne contient pas de valeur de la forme $(\texttt{K}\ \texttt{v})$. Cette « loi » correspondra à une nouvelle règle d'inférence dans le système de types.

Par ailleurs, nous étendons les « ensembles de contraintes » avec un opérateur de « disjonction ». Pour simplifier, et puisqu'elles seront naturellement générées ainsi par les règles d'inférence, nous manipulerons toujours des contraintes sous leur forme normale conjonctive. Nous définissons donc Ψ représentant une disjonction de contraintes :

$\Psi \ ::= \ \texttt{C}_1 \ \vee \ \ldots \ \vee \ \texttt{C}_n \qquad (\text{avec } n \geqslant 0)$

et redéfinissons Φ comme une conjonction de disjonctions de contraintes :

$$\Phi \ ::= \ \Psi_1 \ \wedge \ \ldots \ \wedge \ \Psi_n \quad \text{(avec } n \geqslant 0 \text{ et où les } \Psi_i \text{ sont non-vides)}$$

À cause de cette nouvelle définition de Φ, il faudra bien entendu adapter les règles de saturation pour qu'elles puissent gérer la présence de disjonctions dans les contraintes. En réalité, toutes les règles d'inférence vont être modifiées par l'ajout d'un Ψ.

3.4 Adaptation des règles d'inférence

Toutes les règles d'inférence comportent maintenant, dans chacune de leurs conclusions et de leurs prémisses, un Ψ représentant une potentielle alternative à la propriété (de la forme $(\mathtt{e} : \alpha)$, $(\tau^l \leqslant \tau^r)$, $(\tau^l \leq \tau^r)$, $(\sigma \leqslant \tau^r)$, $(\tau^l \# \mathtt{K})$ ou $(\tau^l \sharp \ \mathtt{K})$) que la règle a en conclusion.

Nouvelles structures des règles d'inférence

La structure des nouvelles règles d'inférence est la suivante :

- Les règles de typage seront maintenant de la forme :

$$\dfrac{\mathrm{T}\ldots \quad \cdots \quad \cdots \quad \cdots}{\Phi, \Gamma \vdash \Psi \vee \mathtt{e} : \alpha \rhd \Phi'}$$

- Les règles de saturation seront maintenant de la forme :

$$\dfrac{\mathrm{S}\ldots \quad \cdots \quad \cdots \quad \cdots}{\Phi \vdash \Psi \vee \tau^l \leqslant \tau^r \rhd \Phi'} \qquad \dfrac{\mathrm{S}\ldots \quad \cdots \quad \cdots \quad \cdots}{\Phi \vdash \Psi \vee \tau^l \leq \tau^r \rhd \Phi'}$$

$$\dfrac{\mathrm{S}\ldots \quad \cdots \quad \cdots \quad \cdots}{\Phi \vdash \Psi \vee \tau^l \# \mathtt{K} \ \rhd \Phi'} \qquad \dfrac{\mathrm{S}\ldots \quad \cdots \quad \cdots \quad \cdots}{\Phi \vdash \Psi \vee \tau^l \sharp \ \mathtt{K} \ \rhd \Phi'}$$

- Les règles d'instanciation seront maintenant de la forme :

$$\dfrac{\mathrm{I}\ldots \quad \cdots \quad \cdots \quad \cdots}{\Phi \vdash \Psi \vee \sigma \leqslant \tau^r \rhd \Phi'}$$

Principes généraux du système de types

Le principe de manipulation de Ψ est le suivant :

- Pour inférer le schéma de type associé à une expression $\mathtt{e}$, Ψ est initialement vide, tout comme Φ et Γ. Ainsi, l'algorithme d'inférence consiste à générer une variable de type α fraîche, à tenter de construire l'arbre d'inférence calculant Φ au dessus de :

$$\varnothing, \varnothing \vdash \varnothing \vee \mathtt{e} : \alpha \vdash \Phi$$

  et, si l'arbre d'inférence a été construit correctement, à renvoyer le schéma de type $\textsc{gen}(\alpha, \Phi, \varnothing)$.
- Toutes les règles de typage, d'instanciation et d'inférence propagent maintenant le Ψ qu'elles ont dans leur conclusion jusqu'à leurs prémisses.
- Nous ajoutons les règles qui seraient des règles de clash dans le système de base, mais qui ici, lorsque Ψ n'est pas vide, ont Ψ en prémisse.
- Les règles de typage du filtrage de motifs sont modifiées. Ce sont elles qui ajoutent des contraintes alternatives dans Ψ.

Nouvelle règle d'instanciation

La nouvelle règle d'instanciation diffère de celle du système de base sur deux points :

- L'ensemble de contraintes présent dans le schéma de types n'est plus une conjonction de contraintes mais une conjonction de disjonctions de contraintes qui sont propagées dans les prémisses de manière similaire aux contraintes.
- Comme pour les autres règles que nous allons définir, le Ψ présent dans la conclusion est simplement propagé dans les prémisses.

On définit donc la règle d'instanciation suivante :

$$\textsc{Inst}\quad \frac{\mathtt{let}\ \alpha'_1,\ldots,\alpha'_n\ \textit{fresh} \quad \Phi \vdash \Psi \vee \alpha_0[\alpha_i \mapsto \alpha'_i]_{i=1}^n \leqslant \tau^r \rhd \Phi_1 \qquad \Phi_1 \vdash \Psi \vee \Psi_1[\alpha_i \mapsto \alpha'_i]_{i=1}^n \rhd \Phi_2 \quad \cdots \quad \Phi_p \vdash \Psi \vee \Psi_p[\alpha_i \mapsto \alpha'_i]_{i=1}^n \rhd \Phi_{p+1}}{\Phi \vdash \Psi \vee [\ \forall \alpha_1\ \ldots\ \alpha_n\ .\ \alpha_0 \mid \Psi_1 \wedge \cdots \wedge \Psi_p\ ] \leqslant \tau^r \rhd \Phi_{p+1}}$$

Transformation des règles de typage classiques

Les règles de typage autres que les règles de gestion du match sont très similaires à celles du système de base. Le Ψ est simplement propagé à l'identique de la conclusion aux prémisses :

- Le typage des constantes se fait grâce à la règle suivante :

$$\textsc{TConst}\quad \frac{\Phi \vdash \Psi \vee T(\mathtt{c}) \leqslant \alpha \rhd \Phi'}{\Phi, \Gamma \vdash \Psi \vee \mathtt{c} : \alpha \rhd \Phi'}$$

- Le typage de l'application d'une primitive se fait par l'une des règles suivantes en fonction de son arité :

TAppLyPrim1

$$\frac{\texttt{let}\ \alpha_1, \alpha_2\ \textit{fresh} \qquad \Phi \vdash \Psi \vee T(\mathtt{p}^1) \leqslant \alpha_1 \to \alpha_2 \rhd \Phi' \qquad \Phi' \vdash \Psi \vee \alpha_2 \leqslant \alpha \rhd \Phi'' \qquad \Phi'', \Gamma \vdash \Psi \vee \mathtt{e}_1 : \alpha_1 \rhd \Phi'''}{\Phi, \Gamma \vdash \Psi \vee \mathtt{p}^1\ \mathtt{e}_1 : \alpha \rhd \Phi'''}$$

TAppLyPrim2

$$\frac{\begin{array}{c}\texttt{let}\ \alpha_0, \alpha_1, \alpha_2, \alpha_3\ \textit{fresh} \qquad \Phi \vdash \Psi \vee \alpha_0 \leqslant \alpha_2 \to \alpha_3 \rhd \Phi' \\ \Phi' \vdash \Psi \vee \alpha_3 \leqslant \alpha \rhd \Phi'' \qquad \Phi'' \vdash \Psi \vee T(\mathtt{p}^2) \leqslant \alpha_1 \to \alpha_0 \rhd \Phi''' \\ \Phi''', \Gamma \vdash \Psi \vee \mathtt{e}_1 : \alpha_1 \rhd \Phi'''' \qquad \Phi'''', \Gamma \vdash \Psi \vee \mathtt{e}_2 : \alpha_2 \rhd \Phi'''''\end{array}}{\Phi, \Gamma \vdash \Psi \vee \mathtt{p}^2\ \mathtt{e}_1\ \mathtt{e}_2 : \alpha \rhd \Phi'''''}$$

- De la même manière, le typage des variables se fait par :

TVar

$$\frac{\texttt{when}\ \Gamma[\mathtt{x}]\ \textit{defined} \qquad \Phi \vdash \Psi \vee \Gamma[\mathtt{x}] \leqslant \alpha \rhd \Phi'}{\Phi, \Gamma \vdash \Psi \vee \mathtt{x} : \alpha \rhd \Phi'}$$

- Le typage des définitions et applications de fonctions sont gérées ainsi :

TLambda

$$\frac{\texttt{let}\ \alpha_1, \alpha_2\ \textit{fresh} \qquad \Phi, \Gamma \oplus (\mathtt{x}, \alpha_1) \vdash \Psi \vee \mathtt{e} : \alpha_2 \rhd \Phi' \qquad \Phi' \vdash \Psi \vee \alpha_1 \to \alpha_2 \leqslant \alpha \rhd \Phi''}{\Phi, \Gamma \vdash \Psi \vee \lambda\ \mathtt{x}\ .\ \mathtt{e} : \alpha \rhd \Phi''}$$

TApp

$$\frac{\texttt{let}\ \alpha_1, \alpha_2, \alpha_3\ \textit{fresh} \qquad \Phi \vdash \Psi \vee \alpha_1 \leqslant \alpha_2 \to \alpha_3 \rhd \Phi' \qquad \Phi' \vdash \Psi \vee \alpha_3 \leqslant \alpha \rhd \Phi'' \qquad \Phi'', \Gamma \vdash \Psi \vee \mathtt{e}_1 : \alpha_1 \rhd \Phi''' \qquad \Phi''', \Gamma \vdash \Psi \vee \mathtt{e}_2 : \alpha_2 \rhd \Phi''''}{\Phi, \Gamma \vdash \Psi \vee \mathtt{e}_1\ \mathtt{e}_2 : \alpha \rhd \Phi''''}$$

- Le typage des couples et des constructeurs de données est transformé de manière similaire :

TPair

$$\frac{\texttt{let}\ \alpha_1, \alpha_2\ \textit{fresh} \qquad \Phi, \Gamma \vdash \Psi \vee \mathtt{e}_1 : \alpha_1 \rhd \Phi' \qquad \Phi', \Gamma \vdash \Psi \vee \mathtt{e}_2 : \alpha_2 \rhd \Phi'' \qquad \Phi'' \vdash \Psi \vee \alpha_1 \times \alpha_2 \leqslant \alpha \rhd \Phi'''}{\Phi, \Gamma \vdash \Psi \vee (\mathtt{e}_1,\ \mathtt{e}_2) : \alpha \rhd \Phi'''}$$

TConstr

$$\frac{\texttt{let}\ \alpha'\ \textit{fresh} \qquad \Phi, \Gamma \vdash \Psi \vee \mathtt{e} : \alpha' \rhd \Phi' \qquad \Phi' \vdash \Psi \vee \mathtt{K}\ \alpha' \leqslant \alpha \rhd \Phi''}{\Phi, \Gamma \vdash \Psi \vee \mathtt{K}\ \mathtt{e} : \alpha \rhd \Phi''}$$

- De la même manière, le typage de la conditionnelle se fait comme avant en propageant en plus le Ψ à toutes ses prémisses :

$$\textsc{TIf}\quad \frac{\texttt{let}\ \alpha'\ \textit{fresh} \qquad \Phi \vdash \Psi \vee \alpha' \leqslant \texttt{bool} \rhd \Phi' \qquad \Phi', \Gamma \vdash \Psi \vee \texttt{e}_1 : \alpha' \rhd \Phi'' \qquad \Phi'', \Gamma \vdash \Psi \vee \texttt{e}_2 : \alpha \rhd \Phi''' \qquad \Phi''', \Gamma \vdash \Psi \vee \texttt{e}_3 : \alpha \rhd \Phi''''}{\Phi, \Gamma \vdash \Psi \vee \texttt{if}\ \texttt{e}_1\ \texttt{then}\ \texttt{e}_2\ \texttt{else}\ \texttt{e}_3 : \alpha \rhd \Phi''''}$$

- Enfin, la règle de typage du `let` subit la même transformation :

$$\textsc{TLet}\quad \frac{\texttt{let}\ \alpha'\ \textit{fresh} \qquad \Phi, \Gamma \vdash \Psi \vee \texttt{e}_1 : \alpha' \rhd \Phi' \qquad \Phi', \Gamma \oplus (\texttt{x}, \textsc{gen}(\alpha', \Phi', \Gamma)) \vdash \Psi \vee \texttt{e}_2 : \alpha \rhd \Phi''}{\Phi, \Gamma \vdash \Psi \vee \texttt{let}\ \texttt{x} = \texttt{e}_1\ \texttt{in}\ \texttt{e}_2 : \alpha \rhd \Phi''}$$

Nouvelles règles de typage du filtrage

Le filtrage de motifs sur les variants polymorphes est maintenant géré par les deux règles suivantes :

$$\textsc{TMatch}\quad \frac{\begin{array}{c}\texttt{let}\ \alpha_e, \alpha_1, \ldots, \alpha_n\ \textit{fresh} \\ \Phi \vdash \Psi \vee \alpha_e \leqslant \{\, \texttt{K}_1\ \alpha_1 \parallel \ldots \parallel \texttt{K}_n\ \alpha_n \,\} \rhd \Phi_0 \qquad \Phi_0, \Gamma \vdash \Psi \vee \texttt{e} : \alpha_e \rhd \Phi_1 \\ \Phi_1, \Gamma \oplus (\texttt{x}_1,\ \alpha_1) \vdash \Psi \vee \alpha_e\ \#\ \texttt{K}_1 \vee \texttt{e}_1 : \alpha \rhd \Phi_2 \\ \ldots \\ \Phi_n, \Gamma \oplus (\texttt{x}_n,\ \alpha_n) \vdash \Psi \vee \alpha_e\ \#\ \texttt{K}_n \vee \texttt{e}_n : \alpha \rhd \Phi_{n+1}\end{array}}{\Phi, \Gamma \vdash \Psi \vee \texttt{match}\ \texttt{e}\ \texttt{with}\ \texttt{K}_1\ \texttt{x}_1 \to \texttt{e}_1 \parallel \ldots \parallel \texttt{K}_n\ \texttt{x}_n \to \texttt{e}_n : \alpha \rhd \Phi_{n+1}}$$

$$\textsc{TMatchDefault}\quad \frac{\begin{array}{c}\texttt{let}\ \alpha_e, \alpha_1, \ldots, \alpha_n, \alpha_d\ \textit{fresh} \\ \Phi \vdash \Psi \vee \alpha_e \leqslant \{\, \texttt{K}_1\ \alpha_1 \parallel \ldots \parallel \texttt{K}_n\ \alpha_n \parallel \alpha_d \,\} \rhd \Phi_0 \qquad \Phi_0, \Gamma \vdash \Psi \vee \texttt{e} : \alpha_e \rhd \Phi_1 \\ \Phi_1, \Gamma \oplus (\texttt{x}_1,\ \alpha_1) \vdash \Psi \vee \alpha_e\ \#\ \texttt{K}_1 \vee \texttt{e}_1 : \alpha \rhd \Phi_2 \\ \ldots \\ \Phi_n, \Gamma \oplus (\texttt{x}_n,\ \alpha_n) \vdash \Psi \vee \alpha_e\ \#\ \texttt{K}_n \vee \texttt{e}_n : \alpha \rhd \Phi_{n+1} \\ \Phi_{n+1}, \Gamma \oplus (\texttt{x}_d,\ \alpha_d) \vdash \Psi \vee \alpha_e \leqslant \{\, \texttt{K}_1\ \alpha_1 \parallel \ldots \parallel \texttt{K}_n\ \alpha_n \,\} \vee \texttt{e}_d : \alpha \rhd \Phi_{n+2}\end{array}}{\Phi, \Gamma \vdash \Psi \vee \texttt{match}\ \texttt{e}\ \texttt{with}\ \texttt{K}_1\ \texttt{x}_1 \to \texttt{e}_1 \parallel \ldots \parallel \texttt{K}_n\ \texttt{x}_n \to \texttt{e}_n \parallel \texttt{x}_d \to \texttt{e}_d : \alpha \rhd \Phi_{n+2}}$$

En dehors du fait qu'elles propagent, comme les autres règles de ce nouveau système, le Ψ de la conclusion aux prémisses ; ces nouvelles règles introduisent dans le Ψ de chaque prémisse correspondant au typage des corps des branches du filtrage ($\texttt{K}_i\ \texttt{x}_i \to \texttt{e}_i$) la nouvelle contrainte alternative « $\alpha_e\ \#\ \texttt{K}_i$ ». Cette contrainte sera alors propagée dans tous les noeuds du sous-arbre d'inférence de $\texttt{e}_i$.

La dernière prémisse de la règle TMatchDefault gérant le cas par défaut est également intéressante. Elle ajoute dans Ψ la contrainte alternative $(\alpha_e \leqslant \{\, \mathtt{K}_1\ \alpha_1 \parallel \ \ldots \ \parallel \ \mathtt{K}_n\ \alpha_n \,\})$ où le variant est ici fermé. Ainsi, toutes les contraintes engendrées par le typage de $(e_d : \alpha)$ seront dans une disjonction faisant apparaître $(\alpha_e \leqslant \{\, \mathtt{K}_1\ \alpha_1 \parallel \ \ldots \ \parallel \ \mathtt{K}_n\ \alpha_n \,\})$. Par conséquent, si le filtrage s'effectue sur l'un des K_i mentionné dans le pattern, aucune des contraintes consécutive à $(e_d : \alpha)$ n'aura besoin d'être vraie. En revanche, si le filtrage s'effectue sur un constructeur qui n'est pas l'un des K_i mentionné dans les filtres, ou sur une valeur qui n'est pas un constructeur, la contrainte $(\alpha_e \leqslant \{\, \mathtt{K}_1\ \alpha_1 \parallel \ \ldots \ \parallel \ \mathtt{K}_n\ \alpha_n \,\})$ sera « invalidée » et toutes les contraintes engendrées par $(e_d : \alpha)$ devront être vérifiées.

Nouvelles règles de saturation

Comme nous l'avons dit précédemment, les règles de saturation vont elles aussi propager la disjonction de contraintes Ψ :

- Les règles permettant le calcul de point fixe lors de la saturation de Φ sont les suivantes :

$$\textsc{SNewConstraint}(\leqslant)\quad \frac{\texttt{when}\ (\Psi \vee \tau^l \leqslant \tau^r) \notin \Phi \qquad \Phi \wedge (\Psi \vee \tau^l \leqslant \tau^r) \vdash \Psi \vee \tau^l \leq \tau^r \rhd \Phi'}{\Phi \vdash \Psi \vee \tau^l \leqslant \tau^r \rhd \Phi'}$$

$$\textsc{SNewConstraint}(\#)\quad \frac{\texttt{when}\ (\Psi \vee \tau^l \,\#\, \mathtt{K}) \notin \Phi \qquad \Phi \wedge (\Psi \vee \tau^l \,\#\, \mathtt{K}) \vdash \Psi \vee \tau^l \,\sharp\, \mathtt{K} \rhd \Phi'}{\Phi \vdash \Psi \vee \tau^l \,\#\, \mathtt{K} \rhd \Phi'} \qquad \textsc{SAlreadyProved}(\leqslant)\quad \frac{\texttt{when}\ (\Psi \vee \tau^l \leqslant \tau^r) \in \Phi}{\Phi \vdash \Psi \vee \tau^l \leqslant \tau^r \rhd \Phi}$$

$$\textsc{SAlreadyProved}(\#)\quad \frac{\texttt{when}\ (\Psi \vee \tau^l \,\#\, \mathtt{K}) \in \Phi}{\Phi \vdash \Psi \vee \tau^l \,\#\, \mathtt{K} \rhd \Phi}$$

  Ainsi, lorsqu'une nouvelle disjonction de contraintes $\Psi \vee \mathtt{C}$ est générée (par l'application d'une règle de typage, d'instanciation ou de saturation), si elle est déjà présente dans Φ, on ne fait rien. Si elle n'a pas encore été ajoutée à Φ, on l'y ajoute et on impose de prouver qu'elle est cohérente avec les autres disjonctions de contraintes présentes dans Φ via l'une des règles suivantes.

- La comparaison entre deux instances de types paramétrés est gérée par la règle suivante. Il s'agit d'une simple adaptation de la règle du système de base dans laquelle le Ψ de la conclusion est propagé dans toutes les prémisses :

$$\textsc{STypeConstr}\quad \frac{\begin{array}{c}\Phi_1 \vdash \Psi \vee \alpha_1 \leqslant \alpha'_1 \rhd \Phi'_1 \qquad \Phi'_1 \vdash \Psi \vee \alpha'_1 \leqslant \alpha_1 \rhd \Phi_2 \\ \ldots \\ \Phi_n \vdash \Psi \vee \alpha_n \leqslant \alpha'_n \rhd \Phi'_n \qquad \Phi'_n \vdash \Psi \vee \alpha'_n \leqslant \alpha_n \rhd \Phi_{n+1}\end{array}}{\Phi_1 \vdash \Psi \vee (\alpha_1, \ldots, \alpha_n)\ t \leq (\alpha'_1, \ldots, \alpha'_n)\ t \rhd \Phi_{n+1}}$$

On y adjoint une règle de saturation utilisée lors d'un clash entre constructeurs de types et reportant la saturation sur Ψ lorsqu'il n'est pas vide :

$$\textsc{STypeConstrClash}\quad \frac{\texttt{when}\ t \neq u \qquad \Phi \vdash \Psi \vee \mathtt{C} \rhd \Phi'}{\Phi \vdash \Psi \vee \mathtt{C} \vee (\alpha_1, \ldots, \alpha_n)\ t \leq (\alpha'_1, \ldots, \alpha'_{n'})\ u \rhd \Phi'}$$

- De manière similaire, le sous-typage sur les variants polymorphes est régi par les règles suivantes :

$$\textsc{SVariantMatch}\quad \frac{\Phi \vdash \Psi \vee \alpha \leqslant \alpha' \rhd \Phi'}{\Phi \vdash \Psi \vee \mathtt{K}\,\alpha \leq \{\ \ldots\ \|\ \mathtt{K}\,\alpha'\ \|\ \ldots\ \} \rhd \Phi'}$$

$$\textsc{SVariantDefault}\quad \frac{\texttt{when}\ \forall i \in [\,1;\ n\,] \mid \mathtt{K} \neq \mathtt{K}_i \qquad \Phi \vdash \Psi \vee \mathtt{K}\,\alpha \leqslant \alpha_d \rhd \Phi'}{\Phi \vdash \Psi \vee \mathtt{K}\,\alpha \leq \{\ \mathtt{K}_1\,\alpha_1\ \|\ \ldots\ \|\ \mathtt{K}_n\,\alpha_n\ \|\ \alpha_d\ \} \rhd \Phi'}$$

$$\textsc{SConstrDefault}\quad \frac{\Phi \vdash \Psi \vee (\alpha_1, \ldots, \alpha_p)\ t \leqslant \alpha'_d \rhd \Phi'}{\Phi \vdash \Psi \vee (\alpha_1, \ldots, \alpha_p)\ t \leq \{\ \mathtt{K}_1\,\alpha'_1\ \|\ \ldots\ \|\ \mathtt{K}_n\,\alpha'_n\ \|\ \alpha'_d\ \} \rhd \Phi'}$$

Lors d'un clash avec un variant sans cas par défaut, on déporte, tout comme via la règle STypeConstrClash, la saturation sur Ψ :

$$\textsc{SVariantMatchClash}\quad \frac{\texttt{when}\ \forall i \in [\,1;\ n\,] \mid \mathtt{K} \neq \mathtt{K}_i \qquad \Phi \vdash \Psi \vee \mathtt{C} \rhd \Phi'}{\Phi \vdash \Psi \vee \mathtt{C} \vee \mathtt{K}\,\alpha \leq \{\ \mathtt{K}_1\,\alpha_1\ \|\ \ldots\ \|\ \mathtt{K}_n\,\alpha_n\ \} \rhd \Phi'}$$

$$\textsc{SConstrMatchClash}\quad \frac{\Phi \vdash \Psi \vee \mathtt{C} \rhd \Phi'}{\Phi \vdash \Psi \vee \mathtt{C} \vee (\alpha_1, \ldots, \alpha_p)\ t \leq \{\ \mathtt{K}_1\,\alpha'_1\ \|\ \ldots\ \|\ \mathtt{K}_n\,\alpha'_n\ \} \rhd \Phi'}$$

- Une variable de type est toujours plus petite qu'elle-même :

$$\textsc{SSameVar}\quad \frac{}{\Phi \vdash \Psi \vee \alpha \leq \alpha \rhd \Phi}$$

- La gestion de la transitivité de $(\leqslant)$ est plus subtile dans ce nouveau système de types. En effet, les fonctions LEFTS et RIGHTS doivent être redéfinies pour prendre en compte la présence de disjonctions dans Φ :

$$\begin{aligned} \textsc{lefts}(\alpha, \Phi) &= \{\ (\Psi, \tau^l) \mid (\Psi \vee \tau^l \leqslant \alpha) \in \Phi\ \} \\ \textsc{rights}(\alpha, \Phi) &= \{\ (\Psi, \tau^r) \mid (\Psi \vee \alpha \leqslant \tau^r) \in \Phi\ \} \end{aligned}$$

De plus, nous définissons une nouvelle fonction (que nous noterons DIFFS) permettant d'extraire de Φ tous les constructeurs de données différents d'une variable de type :

$$\textsc{diffs}(\alpha, \Phi) = \{\ (\Psi, \mathtt{K}) \mid (\Psi \vee \alpha \,\#\, \mathtt{K}) \in \Phi\ \}$$

Il y a en réalité une légère ambiguïté dans la définition de ces fonctions qu'il est nécessaire de préciser. En effet, une première définition pertinente de ces fonctions consiste à rechercher le motif quelle que soit sa position dans les disjonctions. Les Ψ sont alors considérés « à permutation des contraintes près ». Une autre définition valide de ces fonctions consiste à rechercher le motif uniquement à un endroit précis des disjonctions, par exemple à la fin. Malgré le fait que la forme globale des arbres d'inférence est impactée par ce choix de définition, ces deux définitions engendrent en réalité des systèmes de types « équivalents », au sens où ils acceptent exactement le même ensemble de programmes. Néanmoins, les performances de l'implémentation d'un tel système de types varient dans des proportions considérables en fonction de la définition choisie, et ce de manière un peu contre-intuitive. Ce point sera discuté dans le chapitre 6.

Les nouvelles règles gérant l'ajout d'une contrainte de comparaison entre une variable de type et un type sont également compliquées par l'ajout de la nouvelle relation (#). En effet, lors de l'ajout d'une contrainte de la forme $(\tau^l \leqslant \alpha)$, il ne suffit plus d'extraire de Φ tous les τ^r vérifiant $(\alpha \leqslant \tau^r)$ et d'engendrer les contraintes de la forme $\tau^l \leqslant \tau^r$, il faut maintenant également considérer tous les $\mathtt{K}$ tels que $(\alpha \,\#\, \mathtt{K}) \in \Phi$ et engendrer les contraintes de la forme $(\tau^l \,\#\, \mathtt{K})$. Ceci peut se formaliser grâce aux trois règles suivantes :

STRANSRIGHT

$$\frac{\begin{array}{c}\texttt{when } \tau^l \notin \{\ \alpha\ \} \\ \texttt{let } (\Psi_1, \tau_1^r), \ldots, (\Psi_n, \tau_n^r) = \textsc{rights}(\alpha,\ \Phi_1) \qquad \texttt{let } (\Psi'_1, \mathtt{K}_1), \ldots, (\Psi'_p, \mathtt{K}_p) = \textsc{diffs}(\alpha,\ \Phi_1) \\ \Phi_1 \vdash \Psi \vee \Psi_1 \vee \tau^l \leqslant \tau_1^r \rhd \Phi_2 \qquad \cdots \qquad \Phi_n \vdash \Psi \vee \Psi_n \vee \tau^l \leqslant \tau_n^r \rhd \Phi_{n+1} \\ \Phi_{n+1} \vdash \Psi \vee \Psi'_1 \vee \tau^l \,\#\, \mathtt{K}_1 \rhd \Phi_{n+2} \qquad \cdots \qquad \Phi_{n+p} \vdash \Psi \vee \Psi'_p \vee \tau^l \,\#\, \mathtt{K}_p \rhd \Phi_{n+p+1}\end{array}}{\Phi_1 \vdash \Psi \vee \tau^l \leq \alpha \rhd \Phi_{n+p+1}}$$

STRANSLEFT

$$\frac{\begin{array}{c}\texttt{when } \tau^r \notin \{\ \alpha\ \} \qquad \texttt{let } (\Psi_1, \tau_1^l), \ldots, (\Psi_n, \tau_n^l) = \textsc{lefts}(\alpha,\ \Phi_1) \\ \Phi_1 \vdash \Psi \vee \Psi_1 \vee \tau_1^l \leqslant \tau^r \rhd \Phi_2 \qquad \cdots \qquad \Phi_n \vdash \Psi \vee \Psi_n \vee \tau_n^l \leqslant \tau^r \rhd \Phi_{n+1}\end{array}}{\Phi_1 \vdash \Psi \vee \alpha \leq \tau^r \rhd \Phi_{n+1}}$$

STransLeftRight

$$\texttt{when } \alpha_1 \neq \alpha_2$$

$$\begin{array}{l}
\texttt{let } (\Psi_1, \tau_1^l), \ldots, (\Psi_n, \tau_n^l) = \textsc{lefts}(\alpha_1,\ \Phi_{1,1}) \\
\texttt{let } (\Psi'_1, \tau_1^r), \ldots, (\Psi'_p, \tau_p^r) = \textsc{rights}(\alpha_2,\ \Phi_{1,1}) \\
\texttt{let } (\Psi''_1, \mathtt{K}_1), \ldots, (\Psi''_r, \mathtt{K}_r) = \textsc{diffs}(\alpha_2,\ \Phi_{1,1})
\end{array}$$

$$\begin{array}{c}
\Phi_{1,1} \vdash \Psi \vee \Psi_1 \vee \Psi'_1 \vee \tau_1^l \leqslant \tau_1^r \rhd \Phi_{1,2} \ \cdots\ \Phi_{1,p} \vdash \Psi \vee \Psi_1 \vee \Psi'_p \vee \tau_1^l \leqslant \tau_p^r \rhd \Phi_{2,1} \\
\vdots \qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
\Phi_{n,1} \vdash \Psi \vee \Psi_n \vee \Psi'_1 \vee \tau_n^l \leqslant \tau_1^r \rhd \Phi_{n,2} \ \cdots\ \Phi_{n,p} \vdash \Psi \vee \Psi_n \vee \Psi'_p \vee \tau_n^l \leqslant \tau_p^r \rhd \Phi_{n+1,1} \\
\\
\Phi_{n+1,1} \vdash \Psi \vee \Psi_1 \vee \Psi'_1 \vee \tau_1^l \,\#\, \mathtt{K}_1 \rhd \Phi_{n+1,2} \ \cdots\ \Phi_{n+1,r} \vdash \Psi \vee \Psi_1 \vee \Psi'_r \vee \tau_1^l \,\#\, \mathtt{K}_r \rhd \Phi_{n+2,1} \\
\vdots \qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
\Phi_{2n,1} \vdash \Psi \vee \Psi_n \vee \Psi'_1 \vee \tau_n^l \,\#\, \mathtt{K}_1 \rhd \Phi_{2n,2} \quad \cdots \quad \Phi_{2n,r} \vdash \Psi \vee \Psi_n \vee \Psi'_r \vee \tau_n^l \,\#\, \mathtt{K}_r \rhd \Phi_{2n+1,1} \\
\hline
\Phi_{1,1} \vdash \Psi \vee \alpha_1 \leq \alpha_2 \rhd \Phi_{2n+1,1}
\end{array}$$

Comme les (#) se propagent dans l'ordre décroissant des types, la règle STransLeft est beaucoup moins impactée par l'ajout de cette nouvelle relation (#) que STransRight et STransLeftRight.

Il ne reste plus qu'à définir les règles de gestion d'une nouvelle relation ($\tau^l \,\#\, \mathtt{K}$). Pour ce faire, nous définissons les règles suivantes dont l'usage est défini par la structure de τ^l :

- Les constructeurs de type sont toujours différents d'un variant :

SConstrDiff

$$\frac{}{\Phi \vdash \Psi \vee (\alpha_1, \ldots, \alpha_n)\ t \sharp \mathtt{K} \rhd \Phi}$$

- Deux variants de noms différents vérifient la relation (#) sans autre condition :

SDataDiff

$$\frac{\texttt{when } \mathtt{K} \neq \mathtt{K}'}{\Phi \vdash \Psi \vee \mathtt{K}\ \alpha \sharp \mathtt{K}' \rhd \Phi}$$

- Deux variants de même nom ne vérifient pas la relation (#), la saturation est reportée sur Ψ lorsqu'il n'est pas vide :

SDataDiffClash

$$\frac{\Phi \vdash \Psi \vee \mathtt{C} \rhd \Phi'}{\Phi \vdash \Psi \vee \mathtt{C} \vee \mathtt{K}\ \alpha \sharp \mathtt{K} \rhd \Phi'}$$

- Enfin, une nouvelle contrainte de la forme $\alpha \# \mathtt{K}$ est reportée sur tous les τ^l plus petits que α dans Φ :

$$\textsc{SAlphaDiff}\quad \frac{\begin{array}{c}\mathtt{let}\ (\Psi_1, \tau_1^l), \ldots, (\Psi_n, \tau_n^l) = \textsc{lefts}(\alpha, \Phi_1) \\ \Phi_1 \vdash \Psi \vee \Psi_1 \vee \tau_1^l \# \mathtt{K} \rhd \Phi_2 \qquad \cdots \qquad \Phi_n \vdash \Psi \vee \Psi_n \vee \tau_n^l \# \mathtt{K} \rhd \Phi_{n+1}\end{array}}{\Phi_1 \vdash \Psi \vee \alpha \sharp \mathtt{K} \rhd \Phi_{n+1}}$$

Comme avec le système de base, étant donné une expression `e` et une variable de type fraîche α, ces règles d'inférence peuvent être utilisées pour tenter de construire un arbre d'inférence au dessus de :

$$\varnothing, \varnothing \vdash \varnothing \vee \mathtt{e} : \alpha \rhd \Phi$$

Lorsque la construction de cet arbre est possible, l'expression `e` est dite « typable » et un ensemble de disjonctions de contraintes Φ est généré. Le schéma de type inféré pour `e` est alors $\textsc{gen}(\alpha, \Phi, \varnothing)$.

3.5 Exemple simple d'utilisation

Pour bien comprendre le fonctionnement de ce nouveau système de types, et en particulier sa capacité à accepter un code comme :

```
let f = λ x . match x with
  ‖ I → 42
  ‖ S → "quarante deux" in
f I + 1
```

déroulons le typage de ce petit programme.

L'arbre d'inférence complet dérivé au dessus de :

$$\varnothing, \varnothing \vdash \varnothing \vee \ \mathtt{let\ f = \lambda\ x\ .\ match\ x\ with\ I \to 42 \parallel\ S \to "..."\ in\ f\ I + 1}\ :\ \alpha \rhd \Phi$$

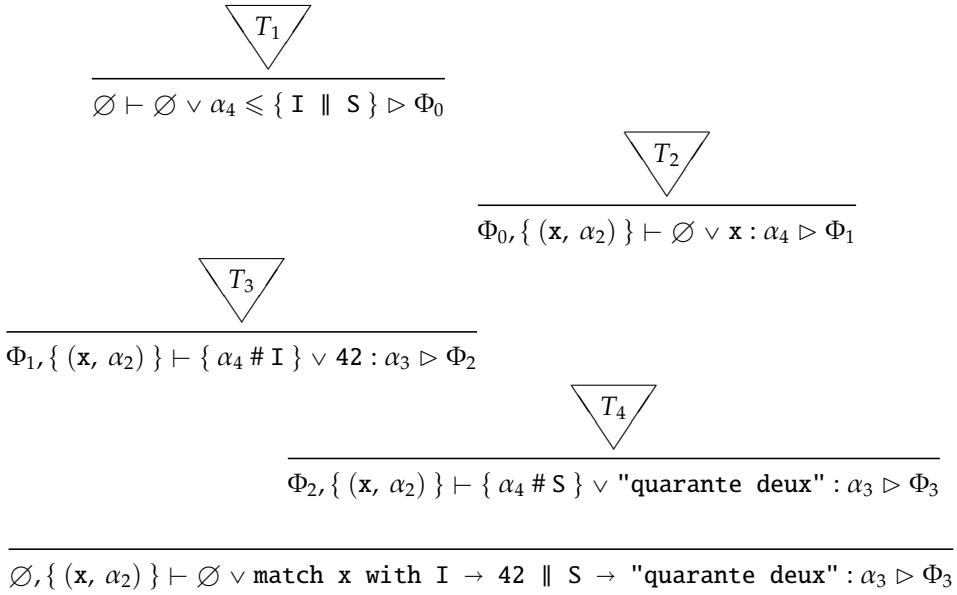
est néanmoins assez gros. Nous ne déroulerons ici que les sous-arbres intéressants dans le contexte de ce nouveau système de types.

L'application des règles de typage TLet et TLambda génèrent trois variables de type fraîches :

- α_1 utilisée pour typer la fonction
- α_2 utilisée pour le type du paramètre `x` de la fonction
- α_3 utilisée pour le type du corps de la fonction

Ces trois variables devront en particulier vérifier la contrainte $(\alpha_2 \to \alpha_3 \leqslant \alpha_1)$ d'après la règle TLambda.

Le sous-arbre correspondant au typage du corps de la fonction est alors de la forme suivante :



dans lequel :

- Le sous-arbre T_1 ne fait qu'ajouter la contrainte :

$$\alpha_4 \leqslant \{\ \mathtt{I} \parallel \mathtt{S}\ \}$$

- Le sous-arbre T_2 correspond à l'instanciation de `x` avec α_2 et engendre la contrainte :

$$\alpha_2 \leqslant \alpha_4$$

- Le sous-arbre T_3 correspond au typage de la constante `42` et engendre la contrainte :

$$\alpha_4 \# \mathtt{I} \vee \mathtt{int} \leqslant \alpha_3$$

  Comme $(\alpha_2 \leqslant \alpha_4) \in \Phi_1$, l'application de la règle SAlphaDiff dans T_3 sur cette nouvelle contrainte génère :

$$\alpha_2 \# \mathtt{I} \vee \mathtt{int} \leqslant \alpha_3$$

- Le sous-arbre T_4 correspond au typage de la constante `"quarante deux"` et engendre la contrainte :

$$\alpha_4 \# \mathtt{S} \vee \mathtt{string} \leqslant \alpha_3$$

  Comme $(\alpha_2 \leqslant \alpha_4) \in \Phi_2$, l'application de la règle SAlphaDiff dans T_4 sur cette nouvelle contrainte génère également :

$$\alpha_2 \# \mathtt{S} \vee \mathtt{string} \leqslant \alpha_3$$

Au final, le schéma de type inféré pour `f` après saturation et suppression, par un algorithme de nettoyage des schémas après généralisation, de la variable α_4 et de ses contraintes associées devenues inutiles est :

$$\begin{aligned}\sigma \;=\; [\,\forall\, \alpha_1\, \alpha_2\, \alpha_3 \;.\; \alpha_1 \;\mid\; & \\ & \alpha_2 \to \alpha_3 \;\leqslant\; \alpha_1 \\ \wedge\; & \alpha_2 \;\leqslant\; \{\, \texttt{I} \parallel \texttt{S} \,\} \\ \wedge\; & (\alpha_2 \,\#\, \texttt{I} \;\vee\; \texttt{int} \leqslant \alpha_3) \\ \wedge\; & (\alpha_2 \,\#\, \texttt{S} \;\vee\; \texttt{string} \leqslant \alpha_3)\,]\end{aligned}$$

dans lequel le lien entre le paramètre (représenté par α_2) et le résultat (représenté par α_3) est explicite.

Grâce à ce schéma de type précis, le typage de l'expression `(f I + 1)` est maintenant possible. En effet, le typage de l'application de l'opérateur $(+)$ via la règle TAppPrim2 va imposer de typer `(f I)` avec une variable de type fraîche α_5 devant vérifier $(\alpha_5 \leqslant \texttt{int})$. Le sous-arbre de typage de `(f I)` sera donc de la forme :

$$\textsc{TApp}\;\dfrac{\dfrac{T_5}{\Phi_4 \vdash \varnothing \vee \alpha_6 \leqslant \alpha_7 \to \alpha_8 \rhd \Phi_5} \qquad \dfrac{T_6}{\Phi_5 \vdash \varnothing \vee \alpha_8 \leqslant \alpha_5 \rhd \Phi_6} \qquad \dfrac{T_7}{\Phi_6, \{\, (\texttt{f},\, \sigma) \,\} \vdash \varnothing \vee \texttt{f} : \alpha_6 \rhd \Phi_7} \qquad \dfrac{T_8}{\Phi_7, \{\, (\texttt{f},\, \sigma) \,\} \vdash \varnothing \vee \texttt{I} : \alpha_7 \rhd \Phi_8}}{\Phi_4, \{\, (\texttt{f},\, \sigma) \,\} \vdash \varnothing \vee \texttt{f I} : \alpha_5 \rhd \Phi_8}$$

dans lequel :

- Les sous-arbres T_5 et T_6 ajoutent simplement les contraintes $(\alpha_6 \leqslant \alpha_7 \to \alpha_8)$ et $(\alpha_8 \leqslant \alpha)$ à l'ensemble de contraintes.
- Le sous-arbre T_7 engendre la contrainte $\sigma \leqslant \alpha_6$. La règle Inst provoque alors une instanciation de σ générant trois nouvelles variables α'_1, α'_2 et α'_3 vérifiant les contraintes :
  - $\alpha'_2 \to \alpha'_3 \leqslant \alpha'_1$
  - $\alpha'_2 \;\leqslant\; \{\, \texttt{I} \parallel \texttt{S} \,\}$
  - $(\alpha'_2 \,\#\, \texttt{I} \;\vee\; \texttt{int} \leqslant \alpha'_3)$
  - $(\alpha'_2 \,\#\, \texttt{S} \;\vee\; \texttt{string} \leqslant \alpha'_3)$
  - $\alpha'_1 \leqslant \alpha_6$

  et déclenchant une cascade de règles de saturation à cause de la présence des contraintes $(\alpha_6 \leqslant \alpha_7 \to \alpha_8)$ et $(\alpha_8 \leqslant \alpha_5)$ dans Φ_5. Cette partie de la saturation engendre en particulier les contraintes $(\alpha_7 \leqslant \alpha'_2)$ et $(\alpha'_3 \leqslant \alpha_5)$ grâce à STypeConstr. L'arrivée de la contrainte $(\alpha'_3 \leqslant \alpha_5)$ est gérée par la règle STransLeftRight qui engendre en particulier la contrainte $(\alpha'_3 \leqslant \texttt{int})$ puisque la contrainte $(\alpha_5 \leqslant \texttt{int})$ a été imposée par l'usage de l'opérateur `(+)`. Cependant, l'ajout de $(\alpha'_3 \leqslant \texttt{int})$ entre en collision avec la contrainte $(\alpha'_2 \,\#\, \texttt{S} \;\vee\; \texttt{string} \leqslant \alpha'_3)$ via la règle

STransLeft et engendre la disjonction $(\alpha'_2 \,\#\, \texttt{S} \vee \texttt{string} \leqslant \texttt{int})$. La règle STypeConstrClash s'applique alors et engendre la contrainte $(\alpha'_2 \,\#\, \texttt{S})$.

- Le sous-arbre T_8 engendre la contrainte $(\texttt{I} \leqslant \alpha_7)$ déclenchant également une cascade de règles de saturation à cause de la présence des contraintes $(\alpha_7 \leqslant \alpha'_2)$ et $(\alpha'_2 \,\#\, \texttt{I} \vee \texttt{int} \leqslant \alpha'_3)$ dans l'ensemble de contraintes Φ_6. L'application de la règle STransRight dans T_7 engendre alors la disjonction $(\texttt{I} \,\#\, \texttt{I} \vee \texttt{int} \leqslant \alpha'_3)$ qui est gérée par la règle SDataDiffClash et engendre la contrainte $(\texttt{int} \leqslant \alpha'_3)$. Comme l'ensemble de contraintes contient également $(\alpha'_3 \leqslant \alpha_5)$ et $(\alpha_5 \leqslant \texttt{int})$, la contrainte $(\texttt{int} \leqslant \texttt{int})$ est générée et gérée par STypeConstr. Aucun clash n'apparaît à ce niveau. Par ailleurs, les contraintes $(\texttt{I} \leqslant \alpha_7)$, $(\alpha_7 \leqslant \alpha'_2)$ et $(\alpha'_2 \,\#\, \texttt{S})$ engendrent la contrainte $(\texttt{I} \,\#\, \texttt{S})$ qui est gérée par la règle SDataDiff et n'engendre pas de clash non-plus.

Au final, la saturation de l'ensemble des contraintes se termine sans erreur et en ayant correctement vérifié la compatibilité entre le `int` généré au typage de la constante `42` et le `int` généré par le typage de l'opérateur `(+)`.

Par ailleurs, l'appel de `f` avec un variant différent de `I` et `S` aurait correctement engendré un clash de typage grâce à la contrainte $(\alpha'_2 \leqslant \{\, \texttt{I} \parallel \texttt{S} \,\})$.

Enfin, l'inversion des motifs `I` et `S` dans le filtrage aurait aussi correctement provoqué un clash de typage car on aurait en particulier abouti aux contraintes :

- $\alpha'_2 \,\#\, \texttt{I} \vee \texttt{string} \leqslant \alpha'_3$ (due au typage du filtrage et à l'instanciation de `f`)
- $\alpha'_3 \leqslant \texttt{int}$ (due à l'application de l'opérateur $(+)$)
- $\texttt{I} \leqslant \alpha'_2$ (due au passage de `I` en argument de `f`)

qui sont incompatibles d'après les règles de saturation que nous avons données. En effet, les deux membres du $(\vee)$ seraient alors invalidés.

3.6 Adaptation de la preuve de terminaison

La preuve de terminaison est quasiment identique à celle du système de types de base. Les deux seules inquiétudes possibles pourraient concerner :

- la présence de la nouvelle relation $(\tau^l \,\#\, \texttt{K})$ dans la grammaire des contraintes. Néanmoins, le nombre de constructeurs de données manipulés par un programme donné est borné, et comme le nombre de types est lui aussi borné dans un sous-arbre de saturation (pour les mêmes raisons qu'avant), le nombre de relations de la forme $(\tau^l \,\#\, \texttt{K})$ est également borné.
- la présence de disjonctions entre les contraintes. Ce n'est pas un problème non plus pour la terminaison de l'algorithme d'inférence car si le nombre de contraintes est borné, le nombre de disjonctions de contraintes est également borné. Il faut tout de même prendre garde dans l'implémentation à empêcher qu'une même contrainte puisse apparaître plusieurs fois dans une même disjonction, et en particulier lorsque l'on effectue un test de la forme $\Psi \in \Phi$.

Pour ce faire, une implémentation naïve pourra maintenir, par exemple, les disjonctions de contraintes sous une forme triée.

Malgré le fait que l'algorithme d'inférence termine toujours, la présence de disjonctions de contraintes pose de gros problèmes de performance en pratique. Une implémentation naïve d'un typeur pour le système présenté dans ce chapitre a été testée et fonctionne suffisamment bien sur de petits exemples en tant que « typeur jouet ». Néanmoins, un tel algorithme ne passe plus du tout à l'échelle alors que c'était le cas pour le système de base, et il est nécessaire de recourir à différentes optimisations pour le rendre utilisable sur de vrais programmes. Ce point est détaillé dans le chapitre 6.

3.7 Adaptation de la preuve de validité

Les principales modifications à apporter dans la preuve de validité concernent, pour la partie typage, les nouvelles règles de typage du filtrage, et pour la partie saturation, la gestion de la nouvelle relation $(\#)$ et de la disjonction. La plupart des lemmes intermédiaires et des preuves correspondantes sont quasiment identiques à ceux du système de types de base.

Mise à jour des définitions de base

La définition de la validité de Φ doit prendre en compte la présence de disjonctions :

Définition 2 (Validité d'un ensemble « Φ » de contraintes)**.**

Un ensemble de contraintes Φ est dit « valide » si toutes ses disjonctions contiennent au moins une contrainte valide.

La définition de la saturation doit prendre en compte la loi associée à la nouvelle relation $(\#)$ ainsi que les disjonctions à propager :

Définition 3 (Saturation d'un ensemble « Φ » de contraintes)**.**

Un ensemble de contraintes Φ est dit « saturé » s'il vérifie toutes les contraintes suivantes :

- *Transitivité de la relation $(\leqslant)$*
  (cf. STransLeft, STransRight, STransLeftRight*) :*
  - $\forall \alpha, \tau^l, \tau^r, \Psi_1, \Psi_2 \ .$
    $(\Psi_1 \vee \tau^l \leqslant \alpha) \in \Phi \wedge (\Psi_2 \vee \alpha \leqslant \tau^r) \in \Phi \Rightarrow$
    $(\Psi_1 \vee \Psi_2 \vee \tau^l \leqslant \tau^r) \in \Phi$
- *Comparaison entre types paramétrés de même nom*
  (cf. STypeConstr*) :*

- $\forall \mathtt{t}, \alpha_1, \ldots, \alpha_n, \alpha'_1, \ldots, \alpha'_n, \Psi$.
  $(\Psi \vee (\alpha_1, \ldots, \alpha_n)\ \mathtt{t} \leqslant (\alpha'_1, \ldots, \alpha'_n)\ \mathtt{t}) \in \Phi \Rightarrow$
  $(\Psi \vee \alpha_1 \leqslant \alpha'_1) \in \Phi \wedge \ldots \wedge (\Psi \vee \alpha_n \leqslant \alpha'_n) \in \Phi \wedge$
  $(\Psi \vee \alpha'_1 \leqslant \alpha_1) \in \Phi \wedge \ldots \wedge (\Psi \vee \alpha'_n \leqslant \alpha_n) \in \Phi$

▶ *Sous-typage sur les variants polymorphes*
(cf. SVariantMatch, SVariantDefault, SConstrDefault*) :*

- $\forall K, \alpha, \alpha', \Psi$.
  $(\Psi \vee K\,\alpha \leqslant \{\ \ldots\ \|\ K\,\alpha'\ \|\ \ldots\ \}) \in \Phi \Rightarrow$
  $(\Psi \vee \alpha \leqslant \alpha') \in \Phi$

- $\forall K, K_1, \ldots, K_n, \alpha, \alpha_d, \alpha_1, \ldots, \alpha_n, \Psi$.
  $\Psi \vee K\,\alpha \leqslant \{\ K_1\ \alpha_1\ \|\ \ldots\ \|\ K_n\ \alpha_n\ \|\ \alpha_d\ \}) \in \Phi \wedge (\forall i\ .\ K \neq K_i) \Rightarrow$
  $(\Psi \vee K\,\alpha \leqslant \alpha_d) \in \Phi$

- $\forall t, \alpha_d, \alpha_1, \ldots, \alpha_n, \Psi$.
  $(\Psi \vee (\alpha_1, \ldots, \alpha_n)\ t \leqslant \{\ \ldots\ \|\ \alpha_d\ \}) \in \Phi \Rightarrow$
  $(\Psi \vee (\alpha_1, \ldots, \alpha_n)\ t \leqslant \alpha_d) \in \Phi$

▶ *Propagation du (#)*
(cf. SAlphaDiff, STransRight, STransLeftRight*) :*

- $\forall \tau^l, \alpha, K, \Psi_1, \Psi_2$.
  $(\Psi_1 \vee \tau^l \leqslant \alpha) \in \Phi \wedge (\Psi_2 \vee \alpha \,\#\, K) \in \Phi \Rightarrow$
  $(\Psi_1 \vee \Psi_2 \vee \tau^l \,\#\, K) \in \Phi$

▶ *En cas d'invalidité d'un membre d'une disjonction, le reste doit être présent dans* Φ
(cf. STypeConstrClash, SVariantMatchClash, SConstrMatchClash, SDataDiffClash*) :*

- $\forall \Psi, \alpha_1, \ldots, \alpha_n, \mathtt{t}, \alpha'_1, \ldots, \alpha'_{n'}, \mathtt{u}$.
  $(\Psi \vee (\alpha_1, \ldots, \alpha_n)\ \mathtt{t} \leqslant (\alpha'_1, \ldots, \alpha'_{n'})\ \mathtt{u}) \in \Phi \Rightarrow$
  $\Psi \in \Phi$

- $\forall \Psi, K, \alpha, K_1, \ldots, K_n, \alpha_1, \ldots, \alpha_n$.
  $(\Psi \vee K\,\alpha \leqslant \{\ K_1\ \alpha_1\ \|\ \ldots\ \|\ K_n\ \alpha_n\ \}) \in \Phi \wedge K \notin \{\ K_i\ \}_{i=1}^n \Rightarrow$
  $\Psi \in \Phi$

- $\forall \Psi, \alpha'_1, \ldots, \alpha'_{n'}, \mathtt{t}, K_1, \ldots, K_n, \alpha_1, \ldots, \alpha_n$.
  $(\Psi \vee (\alpha'_1, \ldots, \alpha'_{n'})\ \mathtt{t} \leqslant \{\ K_1\ \alpha_1\ \|\ \ldots\ \|\ K_n\ \alpha_n\ \}) \in \Phi \Rightarrow$
  $\Psi \in \Phi$

- $\forall \Psi, K, \alpha \ .$
  $(\Psi \vee K\,\alpha \# K) \in \Phi \Rightarrow$
  $\Psi \in \Phi$

Les définitions des relations d'ordre entre ensembles de contraintes et entre schémas de types doivent maintenant prendre en compte la présence des disjonctions :

Définition 12 (Relation d'ordre sur l'ensemble des disjonctions de contraintes (Ψ))**.**

Deux disjonctions de contraintes Ψ et Ψ' vérifient $\Psi \leqslant \Psi'$ s'il existe une fonction totale de renommage R des variables de type de telle sorte que $R(\Psi) \supset \Psi'$.

Définition 5 (Relation d'ordre sur les ensembles de contraintes (Φ))**.**

Deux ensembles de contraintes Φ et Φ' vérifient $\Phi \leqslant \Phi'$ s'il existe une fonction totale de renommage R des variables de type de telle sorte que, pour tout $\Psi \in \Phi$, il existe $\Psi' \in \Phi'$ tel que $R(\Psi) \supset \Psi'$.

L'inclusion des Φ est dans le même sens qu'avant, mais on autorise maintenant les disjonctions à contenir des membres supplémentaires dans le plus petit des ensembles de contraintes. Ces définitions sont adaptées de manière similaire sur les schémas de type et sur les environnements de typage.

La définition de la relation (:) entre une expression et un schéma de type fait maintenant intervenir un Ψ initial vide :

Définition 11 (« $\mathsf{e} \ : \ \sigma$ »)**.**

Une expression e et un schéma de type σ vérifient la relation $\mathsf{e} : \sigma$ si, étant donnée une variable de type fraîche α, les deux propriétés suivantes sont vérifiées :

- $\varnothing, \varnothing \vdash \varnothing \vee \mathsf{e} : \alpha \rhd \Phi$
- $\textsc{gen}(\alpha, \Phi, \varnothing) \leqslant \sigma$

Adaptation des lemmes

Les lemmes de monotonie sont adaptés pour prendre en compte une éventuelle décroissance de la disjonction Ψ :

Lemme 3 (Monotonie de la saturation)**.**

Soient :

- $(\tau^l \leqslant \tau^r)$ *une contrainte de types*
- Φ_1 *et* Φ_2 *deux ensembles de contraintes vérifiant* $\Phi_2 \leqslant_R \Phi_1$
- Ψ_1 *et* Ψ_2 *deux disjonctions de contraintes vérifiant* $\Psi_2 \leqslant_R \Psi_1$
- Φ'_1 *l'ensemble de contraintes obtenu par saturation de* $(\tau^l \leqslant \tau^r)$ *dans* Φ_1 *en dérivant l'arbre d'inférence au dessus de* $\Phi_1 \vdash \Psi_1 \vee \tau^l \leqslant \tau^r \rhd \Phi'_1$.

Alors le déroulage de l'arbre d'inférence au dessus de $\Phi_2 \vdash \Psi_2 \vee \tau^l \leqslant \tau^r \rhd \Phi'_2$ *réussit et génère un ensemble de contraintes* Φ'_2 *vérifiant* $\Phi'_2 \leqslant \Phi'_1$.

Lemme 4 (Monotonie du typage)**.**

Soient :

- e *une expression*
- α *une variable de type*
- R *une fonction de renommage des variables de type*
- Φ_1 *et* Φ_2 *deux ensembles de contraintes vérifiant* $\Phi_2 \leqslant_R \Phi_1$
- Γ_1 *et* Γ_2 *deux environnements de typage vérifiant* $\Gamma_2 \leqslant_R \Gamma_1$
- Ψ_1 *et* Ψ_2 *deux disjonctions de contraintes vérifiant* $\Psi_2 \leqslant \Psi_1$
- Φ'_1 *l'ensemble de contraintes obtenu par typage de* $\mathsf{e} : \alpha$ *dans* Φ_1, Γ_1 *en dérivant l'arbre d'inférence au dessus de* $\Phi_1, \Gamma_1 \vdash \Psi_1 \vee \mathsf{e} : \alpha \rhd \Phi'_1$.

Alors le déroulage de l'arbre d'inférence au dessus de :

$$\Phi_2, \Gamma_2 \vdash \Psi_2 \vee \mathsf{e} : \alpha \rhd \Phi'_2$$

réussit et génère un ensemble de contraintes Φ'_2 *vérifiant* $\Phi'_2 \leqslant \Phi'_1$.

Les démonstrations de ces deux lemmes sont très similaires aux démonstrations correspondantes dans le système de base. Par ailleurs, le fait d'autoriser Ψ à décroître est effectivement utilisé dans les démonstrations des lemmes de réduction du sujet.

Les énoncés des lemmes de réduction du sujet sont très similaires à leurs originaux du système de types de base, mais leurs démonstrations demandent quelques modifications :

Lemme 5 (Réduction du sujet par $(\longrightarrow)$)**.**

Soient :

- e_1 *et* e_2 *deux expressions vérifiant* $e_1 \longrightarrow e_2$
- α *une variable de type*
- Φ *un ensemble de contraintes valide et saturé*
- Φ'_1 *l'ensemble de contraintes obtenu par typage de* $e_1 : \alpha$ *dans* $(\Phi, \varnothing, \varnothing)$ *en dérivant l'arbre d'inférence au dessus de* $\Phi, \varnothing \vdash \varnothing \vee e_1 : \alpha \rhd \Phi'_1$.

Alors le typage de $e_2 : \alpha$ *dans* $(\Phi, \varnothing, \varnothing)$ *réussit et l'ensemble de contraintes* Φ'_2 *obtenu en dérivant l'arbre d'inférence au dessus de* $\Phi, \varnothing \vdash \varnothing \vee e_2 : \alpha \rhd \Phi'_2$ *vérifie* $\Phi'_2 \leqslant \Phi'_1$.

Dans ce lemme, le Ψ est considéré vide, comme l'est Γ. Il ne serait pas utile de généraliser ce théorème à un Ψ non-vide puisque le contexte d'évaluation ne nous autorise jamais à réduire de sous-expression sous un filtrage.

Démonstration (Lemme 5) :

Pour la majorité des cas de réduction par $(\longrightarrow)$, la preuve de ce lemme est très similaire à celle correspondante dans le système de base. La seule différence intéressante (en dehors de la nouvelle gestion du filtrage détaillée ci-après) concerne l'usage de la décroissance de Ψ dans le lemme 4. Cette décroissance est utilisée dans tous les cas de réduction de $(\longrightarrow)$ engendrant un remplacement de la forme `e[x ↦ v]` à savoir, la réduction du `let` d'une valeur, l'application d'un λ a une valeur, et la réduction du filtrage sur une valeur. De telles réductions consistent en un remplacement de toutes les occurrences d'une variable par sa valeur associée dans une expression. La valeur en question est susceptible d'être répliquée n'importe où dans l'expression, y compris dans des cas de filtrage alors qu'elle était auparavant en dehors de tout filtrage. Lorsque cela arrive, la disjonction Ψ présente lors de son typage sera augmentée de nouvelles contraintes alternatives provenant des cas de filtrage dans lesquels elle est rentrée. Cette augmentation potentielle du nombre de membres dans les disjonctions ne pose aucun problème pour la décroissance de Φ à chaque pas d'évaluation (qui est une propriété importante à démontrer pour le lemme actuel) car elle est prise en compte dans la nouvelle définition de la relation d'ordre $(\leqslant)$ sur les ensembles de contraintes Φ.
Les cas subissant de réelles modifications et auxquels nous allons nous intéresser maintenant sont ceux concernant le filtrage :

- Cas « `match K v with ... ‖ K x → e ‖ ...` $\longrightarrow$ `e[x ↦ v]` » :
  L'arbre d'inférence de

$$\Phi, \varnothing \vdash \varnothing \vee \texttt{match K v with ... ‖ K x → e ‖ ... :}\ \alpha \rhd \Phi'_1$$

est de la forme :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Phi \vdash \varnothing \vee \alpha_e \leqslant \{\ \ldots \parallel \mathtt{K}\,\alpha_1 \parallel \ \ldots\} \rhd \Phi_1}\,T_1}{\Phi_2 \vdash \varnothing \vee \alpha_0 \leqslant \alpha_1 \rhd \Phi_3}\,T_3}{\Phi_2 \vdash \varnothing \vee \mathtt{K}\,\alpha_0 \leq \{\ \ldots \parallel \mathtt{K}\,\alpha_1 \parallel \ \ldots\} \rhd \Phi_3}\,\textsc{SVariantMatch}}{\Phi_2 \vdash \varnothing \vee \mathtt{K}\,\alpha_0 \leqslant \{\ \ldots \parallel \mathtt{K}\,\alpha_1 \parallel \ \ldots\} \rhd \Phi_3}\,\textsc{SNewConstraint}}{\Phi_2 \vdash \varnothing \vee \mathtt{K}\,\alpha_0 \leq \alpha_e \rhd \Phi_3}\,\textsc{STransRight}}{\Phi_2 \vdash \varnothing \vee \mathtt{K}\,\alpha_0 \leqslant \alpha_e \rhd \Phi_3}\,\textsc{SNewConstraint}$$

$$\textsc{TConstr}\ \dfrac{\dfrac{}{\Phi_1, \varnothing \vdash \varnothing \vee \mathtt{v} \ :\ \alpha_0 \rhd \Phi_2}\,T_2 \qquad \Phi_2 \vdash \varnothing \vee \mathtt{K}\,\alpha_0 \leqslant \alpha_e \rhd \Phi_3}{\Phi_1, \varnothing \vdash \varnothing \vee \mathtt{K\ v} \ :\ \alpha_e \rhd \Phi_3}$$

$$\textsc{TMatch}\ \dfrac{\dfrac{}{\ldots}\ \ldots \qquad \dfrac{\overset{\mathtt{x}:\alpha' \qquad \mathtt{x}:\alpha''}{\overset{\vee}{\overset{\vee}{\Phi_3, \{\ (\mathtt{x}, \alpha_1)\ \} \vdash \{\ \alpha_e \,\#\, \mathtt{K}\ \} \vee \mathtt{e} : \alpha \rhd \Phi_4}}\,T_4}\ \ldots}{\Phi, \varnothing \vdash \varnothing \vee \mathtt{match\ K\ v\ with\ \ldots \parallel K\ x \to e \parallel\ \ldots} \ :\ \alpha \rhd \Phi_1'}$$

Comme précédemment, la preuve consiste à montrer qu'il est possible de construire un arbre d'inférence au dessus de $\Phi_3, \varnothing \vdash \varnothing \vee \mathtt{e[x \mapsto v]} : \alpha \rhd \Phi_2'$ en remplaçant dans T_4 les sous-arbres concernant les occurrences libres de $\mathtt{x}$ par une adaptation de T_2.

La propriété intéressante à remarquer concerne l'apparition de la contrainte $\{\ \alpha_e \,\#\, \mathtt{K}\ \}$ dans Ψ lors du typage de $(\mathtt{e} : \alpha)$. Cette contrainte alternative est due au nouveau système de types et il faut montrer qu'elle ne casse pas la validité du typage. Néanmoins, elle pourrait être très gênante car elle n'est pas présente dans l'arbre que l'on souhaite construire au dessus de $\Phi_3, \varnothing \vdash \varnothing \vee \mathtt{e[x \mapsto v]} : \alpha \rhd \Phi_2'$.

La raison pour laquelle la validité n'est pas cassée par la présence de cette contrainte alternative est due à la présence de $(\mathtt{K}\ \alpha_0 \leqslant \alpha_e)$ dans Φ_3. Cette contrainte provoque en effet la suppression systématique de $(\alpha_e \,\#\, \mathtt{K})$ dans toutes les disjonctions générées dans T_4. Plus précisément, chaque contrainte générée dans T_4 est initialement de la forme $(\alpha_e \,\#\, \mathtt{K} \vee \mathtt{C})$ puisque $(\alpha_e \,\#\, \mathtt{K})$ est présent dans le Ψ qui est propagé depuis la racine de T_4. À chaque apparition d'une contrainte de cette forme, la règle SAlphaDiff est appliquée et génère la contrainte $(\mathtt{K} \,\#\, \mathtt{K} \vee \mathtt{C})$ à cause de la présence de $(\mathtt{K}\ \alpha_0 \leqslant \alpha_e)$.

Cette nouvelle contrainte est alors gérée par la règle SDataDiffClash qui engendre la contrainte $\mathtt{C}$. Nous sommes alors sûrs que T_4 possède des sous-arbres prouvant chacune des contraintes imposées par le typage de $\mathtt{e}$ à la fois avec et sans la contrainte alternative $(\alpha_e \,\#\, \mathtt{K})$. Le fait que $(\alpha_e \,\#\, \mathtt{K})$ ne soit pas dans le Ψ de l'arbre que l'on souhaite construire au dessus de $\Phi_3, \varnothing \vdash \varnothing \vee \mathtt{e[x \mapsto v]} : \alpha \rhd \Phi'_2$ n'est donc pas un problème.

- Cas « $\mathtt{match\ v\ with\ K_1\ x_1 \to e_1 \parallel \ldots \parallel K_n\ x_n \to e_n \parallel x_d \to e_d} \longrightarrow \mathtt{e_d[x_d \mapsto v]}$ »
  lorsque $\mathtt{v}$ n'est pas de la forme $(\mathtt{K}_i\ \_)$ pour $i \in [\,1;\, n\,]$:
  L'arbre d'inférence est alors de la forme :

$$\textsc{TMatchDefault}\ \dfrac{\dfrac{T_1}{\Phi \vdash \varnothing \vee \alpha_e \leqslant \{\, \mathtt{K}_1\ \alpha_1 \parallel \ldots \parallel \mathtt{K}_n\ \alpha_n \parallel \alpha_d \,\} \rhd \Phi_1} \quad \dfrac{T_2}{\Phi_1, \varnothing \vdash \varnothing \vee \mathtt{v} : \alpha_e \rhd \Phi_2} \quad \dfrac{\dfrac{\ldots}{\ldots} \quad \dfrac{\overset{\mathtt{x}_d : \alpha' \quad \mathtt{x}_d : \alpha''}{T_3}}{\Phi_2, \{\, (\mathtt{x}_d, \alpha_d) \,\} \vdash \{\, \alpha_e \leqslant \{\, \mathtt{K}_1\ \alpha_1 \parallel \ldots \parallel \mathtt{K}_n\ \alpha_n \,\} \,\} \vee \mathtt{e}_d : \alpha \rhd \Phi'_1}}{\Phi, \varnothing \vdash \varnothing \vee \mathtt{match\ v\ with\ K_1\ x_1 \to e_1 \parallel \ldots \parallel K_n\ x_n \to e_n \parallel x_d \to e_d} : \alpha \rhd \Phi'_1}$$

  La gestion de ce cas est similaire.
  Le fait que $\mathtt{v}$ ne soit pas de la forme $(\mathtt{K}_i\ \_)$ nous assure que la contrainte qui est générée par le typage de $(\mathtt{v} : \alpha_e)$ entre en collision avec la contrainte alternative $\alpha_e \leqslant \{\, \mathtt{K}_1\ \alpha_1 \parallel \ldots \parallel \mathtt{K}_n\ \alpha_n \,\}$. Ainsi, de la même manière que dans le cas précédent, nous sommes assurés que toutes les contraintes générées dans T_4 de la forme $(\alpha_e \leqslant \{\, \mathtt{K}_1\ \alpha_1 \parallel \ldots \parallel \mathtt{K}_n\ \alpha_n \,\} \vee \mathtt{C})$ engendrent la preuve de cohérence de la contrainte $\mathtt{C}$ avec le reste de Φ.

Les deux autres lemmes de réduction du sujet et leurs démonstrations correspondantes sont très similaires à celles du système de base. Il s'agit de simples démonstrations par induction qui ne font pas intervenir le typage des cas de filtrage puisque la définition du contexte d'évaluation nous interdit de réduire sous un filtrage.

La démonstration du lemme 8 (« les expressions bloquées ne sont pas typables ») ne change pas en dehors des Ψ qui sont propagés dans les arbres d'inférence. Le langage n'ayant pas changé, tout comme sa sémantique, les expressions bloquées sont les mêmes qu'avant et les raisons pour

lesquelles elles sont refusées par le typeur sont identiques, y compris pour les cas d'expressions bloquées faisant intervenir un filtrage.

Le lemme 9 (« évaluation uniforme ») ne concerne que la sémantique du langage. Il reste inchangé, tout comme sa démonstration.

Enfin, les théorèmes de validité sont des conséquences directes des lemmes précédents. Leurs démonstrations sont identiques.

3.8 Exemple concret d'application : un encodage des « objets »

Une utilisation intéressante de l'extension du filtrage de motifs que nous avons présentée dans ce chapitre est d'encoder un modèle objet sans ajouter la moindre construction supplémentaire à notre langage ni changer sa sémantique.

Idée générale

Le principe de base est simple : un objet est encodé par une fonction attendant en premier argument un variant représentant un nom de méthode. En voici un exemple :

```
let obj = λmeth. match meth with
  ‖ ToString → "<obj>"
  ‖ GetState → Alive in
...
```

Pour appeler une méthode sur un tel objet, il suffit de lui donner le variant correspondant en argument :

```
println (obj ToString)
```

Une méthode peut bien entendu être une valeur fonctionnelle et attendre des arguments supplémentaires comme pour l'objet suivant :

```
let smart_animal = λaction. match action with
  ‖ AddIntegers → λn. λp. n + p
  ‖ NegateABool → λb. if b then false else true
  ‖ Cry         → println "piu piu" in
...
```

sur lequel il est possible d'appeler la méthode `AddIntegers` ainsi :

```
print_int (smart_animal AddIntegers 42 123)
```

Dans le système de types présenté dans ce chapitre, les différents cas du filtrage sont « typés indépendamment », et les contraintes de types engendrées par le typage de chaque cas sont liées au variant passé en argument à l'objet grâce à une disjonction de contraintes. Grâce à cela, les types des différentes méthodes ne sont pas « confondues » lors d'un appel de méthode comme ce serait le cas avec le système de base. C'est grâce à cette propriété qu'il est possible de créer, mais surtout d'utiliser, des objets ayant des méthodes de types différents.

Classes

Une « classe » dans ce modèle objet peut alors être simplement encodée par une fonction renvoyant un objet comme par exemple :

```
let atom = λname. λnum. λmeth. match meth with
  ‖ GetName         → name
  ‖ GetAtomicNumber → num in
...
```

Il est alors possible d'« instancier » cette classe en effectuant un appel partiel :

```
let lawrencium = atom "Lr" 103 in ...
```

Nous remarquerons néanmoins que cette variable `lawrencium` n'est ensuite utilisable dans différents contextes d'appel que si le type qui lui a été associé a bien été généralisé. C'est le cas avec notre système de types actuel puisque notre langage ne possède pas de données mutables. Nous n'avons donc (pour l'instant) pas de contraintes en rapport avec la « value-restriction ». Dans le contexte d'un système de types appliquant une forme de « value-restriction » pour décider quand généraliser, l'instanciation d'une classe serait plus compliquée. Dans ce cas particulier, il suffirait de définir `lawrencium` ainsi :

```
let lawrencium meth = atom "Lr" 103 meth in ...
```

pour assurer la généralisation.

Héritage simple

Pour encoder une forme d'héritage simple, il suffit d'utiliser le cas par défaut du filtrage et de déléguer l'appel de méthode à un autre objet (le « parent », en l'occurrence) comme dans l'exemple suivant :

```
let point = λx. λy. λmeth. match meth with
  ‖ Coord     → (x, y)
  ‖ Norm      → sqrt (x * x + y * y)
  ‖ ClassName → "point" in
```

```
let colored_point = λx. λy. λcolor.
  let super = point x y in
  λmeth. match meth with
    ‖ Color     → color
    ‖ ClassName → "colored_point"
    ‖ _         → super meth in

...
```

Limitations

Cet encodage des objets présente plusieurs limitations importantes :

1. L'accès à « `self` » est compliqué. Une solution pour y accéder est d'encoder les objets par une fonction récursive :

   ```
   let class = λx.
     let rec obj = λmeth. match meth with
       ‖ GetX → x
       ‖ Add  → λy. obj GetX + y in
     obj in
   ...
   ```

   Néanmoins, si l'objet est utilisé plusieurs fois dans sa définition avec des méthodes différentes, on rencontre le problème classique de récursion polymorphe. Avec le système de types actuel, une telle implémentation des objets nous impose que toutes les méthodes utilisées en interne à un objet soient de même type, ce qui n'est pas raisonnable. Ce problème peut être réglé, soit via une annotation de type comme on le fera pour gérer la récursion polymorphe dans le chapitre 5, soit partiellement grâce à l'extension du polymorphisme présentée dans le prochain chapitre.

2. Pour la même raison, la redéfinition de méthodes est un problème, même en cas d'héritage simple. En effet, pour que le parent puisse accéder aux méthodes réelles de l'objet (qui ont potentiellement été surchargées), une technique est de définir les objets de sorte qu'ils se prennent eux même en argument. Un tel encodage serait envisageable de la manière suivante :

   ```
   let point = λx. λy. λself. λmeth. match meth with
     ‖ GetX → x ‖ GetY → y
     ‖ Coord → (self self GetX, self self GetY) in

   let opposite_point = λx. λy.
     let super = point x y in
   ```

```
  λself. λmeth. match meth with
    | GetX → -x | GetY → -y
    | _ → super self meth in

let apoint = opposite_point 3 4 in
apoint apoint Coord
```

Les fonctions définies ci-dessus effectuent de l'auto-application, elles sont actuellement acceptées par le typeur parce que nous n'avons pas interdit les types récursifs dans nos systèmes. Ce code présente malheureusement le même problème de polymorphisme que le précédent, il est impossible actuellement d'appeler des méthodes de types différents dans un même objet. Ce problème pourra être réglé de manière similaire comme nous le verrons par la suite.

3. L'encodage de l'héritage multiple est compliqué. Il serait bien évidemment possible de le simuler en listant toutes les méthodes de ses multiples parents et en effectuant le dispatch manuellement, mais cette technique n'est pas très modulaire. En effet, en cas d'ajout d'une méthode dans une classe, il faudrait modifier toutes ses sous-classes pour mettre à jour les codes de dispatch.
   Une technique beaucoup plus élégante consiste à définir les objets comme des fonctions attendant un troisième argument (en plus de `self` et de `meth`) représentant l'objet à qui déléguer un appel de méthode qu'il ne sait ni gérer lui-même, ni déléguer à l'un de ses parents. Voici un exemple d'un tel encodage :

```
let point = λx. λdeleg. λthis. λmeth. match meth with
  | GetX → x
  | _ → deleg this meth in

let addr = λstreet. λdeleg. λthis. λmeth. match meth with
  | GetStreet → street
  | _ → deleg this meth in

let point_addr = λx. λstreet.
  let super_point = point x in
  let super_address = address street in
  λdeleg. λthis. λmeth. match meth with
    | GetClass → "point_addr"
    | _ → super_address (super_point deleg) this meth in

let root = λthis. λmeth. match meth with
  | ToString -> "<obj>" in
```

```
let o = point_addr 3 "gambetta" in
o root o GetX
```

Cet encodage présente la même limitation qu'avant : un même délégué ne peut pas être appelé avec des méthodes de types différents en argument.

4. Enfin, une limitation majeure est l'impossibilité de définir, indépendamment de toute classe ou objet, une fonction attendant un objet générique en argument et appelant plusieurs méthodes dessus dans des contextes de typage différents. Par exemple, le code suivant

   ```
   let f = λobj. println (obj ToString); obj GetX + 1 in
   f point
   ```

   est refusé. En effet, la variable `obj` est utilisée deux fois dans le corps de `f` avec un type de retour attendu différent. L'appel à `f` dans ce code est alors refusé car toutes les contraintes associées à `obj` sont appliquées sur l'unique variable de type générée par la règle TLambda pour son paramètre. L'objet donné en argument de `f` doit donc être capable de renvoyer des valeurs qui sont à la fois compatibles avec `int` et `string`, ce qui est impossible.

Dans tous les cas, nous nous heurtons au même problème, à savoir l'utilisation impossible dans le corps d'une fonction d'un même paramètre dans des « contextes de typage différents ». L'un des buts de l'extension présentée dans le prochain chapitre est de lever cette limitation.

Chapitre 4

Amélioration de la généralisation

Les deux systèmes de types que nous avons étudiés jusqu'ici présentent un problème classique lié à la généralisation. Ce problème intervient typiquement lorsque l'on souhaite qu'une fonction utilise son argument de manière polymorphe, c'est-à-dire plusieurs fois dans des « contextes de typage » différents. Une telle chose est pour l'instant impossible.

Une approche standard de ce problème est basée sur l'usage de « types conjonctifs ». Bien que très puissante, cette technique pose des problèmes de calculabilité de l'algorithme d'inférence qui la sous-tend.

Nous nous proposons, dans ce chapitre, d'aborder ce problème avec une technique originale consistant à prolonger la durée de vie des schémas de type lors de la saturation de l'ensemble de contraintes. Dans un premier temps, ceci remettra en cause la terminaison de nos algorithmes d'inférence. Nous aurons alors recours à une technique basée sur l'annotation des relations de sous-typage par des clés de taille bornée pour assurer la terminaison de la saturation des contraintes tout en restreignant, de manière paramétrable, la puissance du système de types obtenu.

La technique de généralisation présentée ici sera plus puissante que celle appliquée jusqu'ici sur la construction (`let x =` e_1 `in` e_2`)`. Cette construction de syntaxe, sémantiquement équivalente à ((λ `x .` e_2`)` e_1`)` deviendra alors obsolète dans ce nouveau système car moins générale.

4.1 Contexte

Polymorphisme

Le polymorphisme, qu'il soit paramétrique ou de sous-typage, représente la capacité d'un objet (une fonction, une donnée structurée, . . .) à être utilisé sous des contraintes de typage différentes, potentiellement incompatibles enre elles. C'est le cas du « polymorphisme d'inclusion » (que l'on appelle aussi « polymorphisme de sous-typage ») cher aux langages à objets, qui permet par exemple à une fonction d'accepter des arguments appartenant à un sous-type de son domaine de définition. Le type polymorphe de la fonction en question est en quelque sorte un *schéma* de type, représentant toute une famille de types effectifs.

C'est aussi le cas du polymorphisme dit « paramétrique » de ML (cf. [DM82]), qui attribue aux valeurs définies localement des schémas de type, construits par quantification universelle de

certaines variables de type, et qui représentent tous les types auxquels il est possible d'utiliser ces valeurs locales.

En ML, le typage de la construction ($\texttt{let x} = \texttt{e}_1 \texttt{ in } \texttt{e}_2$), qui introduit et utilise le polymorphisme, peut être vu sous deux angles différents :

- la synthèse pour $\texttt{e}_1$, d'un type universellement quantifié (un schéma), et l'instanciation de ce schéma en des types distincts à chacun des usages de l'identificateur `x` dans $\texttt{e}_2$,
- la synthèse de types classique de $\texttt{e}_2$ dans laquelle chaque occurrence de `x` reçoit un type différent (le type auquel l'occurrence en question est utilisée), et la vérification de la typabilité de $\texttt{e}_1$ par chacun de ces types.

La première vision exhibe un polymorphisme sous la forme de types universellement quantifiés, et s'inscrit dans le champ des travaux autour du λ-calcul polymorphe dont le système F est probablement le représentant le plus connu. La seconde vision, en utilisant un typage spécifique de chacune des occurrences de `x`, se généralise en considérant que la valeur `x` a pour type l'intersection du type de chacune de ses occurrences.

Le polymorphisme permettant le typage unique d'une expression qui pourra ensuite être utilisé à des instances distinctes, offre des qualités appréciables pour le typage de bibliothèques de conteneurs génériques (listes, tableaux, arbres). De nombreux travaux ont donc cherché à étendre les capacités dans certains cas un peu limitées de ML en matière de polymorphisme.

Nous mentionnons ci-dessous quelques-uns de ces travaux, en les classant selon les deux axes ci-dessus : polymorphisme à la système F, et types intersection.

Polymorphisme à la système F

Le système F, introduit indépendamment par Girard et Reynolds (cf. [G86]), est une extension du λ-calcul simplement typé qui autorise l'usage de types universellement quantifiés, les quantificateurs pouvant apparaître à une profondeur arbitraire dans les schémas de type. *A contrario*, la quantification des schémas de types de ML est prénexe : les quantificateurs ne peuvent apparaître qu'en tête d'un schéma. La puissance du polymorphisme du système F a toutefois un prix : l'inférence de types y est indécidable, comme l'a démontré Wells en 1994 (cf. [W96]). Enrichir le polymorphisme simple de ML en direction de celui du système F, tout en préservant la possibilité d'inférence, a représenté une voie de recherches qui a été largement explorée, notamment par Rémy (cf. [CR14]) et Leijen (cf. [L08]).

Types intersection

Le domaine duquel l'extension que nous présentons ici se rapproche le plus est probablement celui des types intersection (cf. [KW99, P91]).

Le principe de base des types intersection consiste à étendre la grammaire des types en ajoutant un opérateur d'intersection noté ($\wedge$). Cet opérateur permet de relier deux types, il ne doit pas

être confondu avec notre opérateur $(\wedge)$ permettant de relier deux contraintes de type. Certaines approches comme celle de Kfoury et Wells (cf. [KW99]) étendent en plus la grammaire des types avec un ensemble de « variables d'expansion », noté F, permettant de conserver l'existence d'un type principal et la décidabilité dans un cadre restreint des types intersections. La grammaire des types (τ) est alors typiquement de la forme suivante :

$$\overline{\tau} \ ::= \ \alpha \mid \tau \to \overline{\tau}$$

$$\tau \ ::= \ \overline{\tau} \mid \tau \wedge \overline{\tau} \mid F \ \tau$$

L'opérateur d'intersection permet en particulier, lorsqu'il lie deux types à gauche d'une flèche, d'imposer que l'argument de la fonction soit à la fois « de deux types distincts ». Il est alors possible de définir des fonctions utilisant leur paramètre dans plusieurs contextes de typage incompatibles. Lors du typage d'une application, l'argument est alors contraint à vérifier chacun des types associés au paramètre de la fonction.

Dans un autre contexte qu'est celui des variants polymorphes à la Garrigue (cf. [G04[1]]), les types intersections jouent également un rôle théorique important. L'opérateur d'intersection $(\wedge)$ entre les types n'est alors autorisé qu'au niveau des arguments des constructeurs de données polymorphes. Il permet, dans certains cas, de retarder un clash de typage dans l'espoir que celui-ci ne se produise jamais. Il arrive en effet que le sous-typage sur les variants polymorphes engendre la « disparition », dans un type, d'un constructeur de données, et donc de toutes les contraintes associées à ses arguments. Cette technique permet en particulier de conserver la principalité du typage, propriété importante de la synthèse de types.

Nous allons, de notre côté, chercher à gagner en polymorphisme de manière un peu similaire en autorisant plusieurs ensembles de contraintes, incompatibles entre eux, à être liés à une même variable de type. Le retard de l'instanciation de certains schémas nous permettra alors de gérer séparément ces ensembles de contraintes, tout en assurant la terminaison du typage.

4.2 Motivation

Considérons les deux exemples de code suivants mettant en évidence une faiblesse classique de nos systèmes de types précédents liée à un manque de généralisation :

Premier exemple

Le premier exemple de code que nous nous proposons d'étudier, sans pour autant être représentatif d'un usage pratique de l'extension que nous présentons dans ce chapitre, a le mérite d'être assez minimaliste et de bien mettre en évidence le problème que nous souhaitons aborder :

```
(λ id . (id 42, id "hello")) (λ x . x)
```

Cet exemple de code est refusé dans un système de types classique à base d'unification et provoque un clash `int`/`string` sur le type du paramètre de `id`.

Néanmoins, le code :

```
let id = λ x . x in (id 42, id "hello")
```

est sémantiquement équivalent et est quant à lui typable dans tout système classique muni d'un mécanisme de généralisation appliquée lors du typage de la construction `let`.

Dans les systèmes de types à base de sous-typage que nous avons étudiés jusqu'ici, le code :

```
(λ id . (id 42, id "hello")) (λ x . x)
```

est accepté mais le schéma de types qui lui est inféré est beaucoup trop restrictif et rend les membres du couple résultat presque inutilisables. En effet, si nous déroulons l'inférence de types via l'un de nos systèmes précédents sur cet exemple, il est facile de vérifier que nous obtenons le schéma de type :

$$[\, \forall \alpha_1 \alpha_2 \alpha_3 \,.\, \alpha_1 \mid \alpha_2 \times \alpha_3 \leqslant \alpha_1 \ \wedge\ \texttt{int} \leqslant \alpha_2 \ \wedge\ \texttt{string} \leqslant \alpha_2 \ \wedge\ \texttt{int} \leqslant \alpha_3 \ \wedge\ \texttt{string} \leqslant \alpha_3 \,]$$

Dans ce schéma, nous avons perdu la connaissance du fait que le premier élément du couple est un entier et le second une chaîne. Par conséquent, pour être déconstruit, un membre du couple résultat doit être utilisé dans un contexte capable d'accepter à la fois un entier et une chaîne de caractère. Ceci impose d'utiliser, soit une primitive générique (comme `print`) ne faisant aucune hypothèse sur la structure de son argument, soit un mécanisme de dispatch dynamique en fonction du type lorsque cela est possible comme il a été mentionné dans le chapitre précédent. Quoi qu'il en soit, cette faiblesse dans les informations de typage inférées est un problème pour l'expressivité du langage. Il serait largement préférable d'inférer le schéma :

$$[\, \forall \alpha_1 \alpha_2 \alpha_3 \,.\, \alpha_1 \mid \alpha_2 \times \alpha_3 \leqslant \alpha_1 \ \wedge\ \texttt{int} \leqslant \alpha_2 \ \wedge\ \texttt{string} \leqslant \alpha_3 \,]$$

qui est lui aussi valide pour cette expression mais beaucoup plus « précis ».

Second exemple

Considérons maintenant un exemple de code, nettement plus intéressant en pratique. Il correspond à l'implémentation des objets présentée dans la dernière section du chapitre précédent :

```
let f = λ obj . println (obj S); obj I + 1 in

let obj = λ meth . match meth with
  ‖ I → 42
  ‖ S → "quarante deux" in

f obj
```

où `println` est une primitive attendant une chaîne de caractères en argument, c'est-à-dire de schéma de type :

$$T(\texttt{println}) \triangleq \{\, \forall \alpha_1\alpha_2\alpha_3 \,.\, \alpha_1 \mid \alpha_2 \to \alpha_3 \leqslant \alpha_1 \ \wedge\ \alpha_2 \leqslant \texttt{string} \ \wedge\ \texttt{unit} \leqslant \alpha_3 \,\}$$

Un tel code est refusé par nos systèmes de types précédents, y compris celui du chapitre 3 possédant un typage affiné du filtrage de motifs. En effet, le schéma inféré pour `f` est alors :

$$\begin{aligned}\texttt{f} : [\, \forall\, \alpha_1\ \alpha_2\ \alpha_3\ \alpha_4\ \alpha_5\ \alpha_6\ \alpha_7\ .\ \alpha_1 \mid\ & \\ & \alpha_2 \to \alpha_3 \leqslant \alpha_1 \ \wedge\ \alpha_2 \leqslant \alpha_4 \to \alpha_5 \ \wedge\ \alpha_2 \leqslant \alpha_6 \to \alpha_7 \\ & \wedge\ \texttt{S} \leqslant \alpha_4 \ \wedge\ \alpha_5 \leqslant \texttt{string} \\ & \wedge\ \texttt{I} \leqslant \alpha_6 \ \wedge\ \alpha_7 \leqslant \texttt{int} \\ & \wedge\ \texttt{int} \leqslant \alpha_3\,]\end{aligned}$$

Par ailleurs, le schéma de type inféré pour `obj` est :

$$\begin{aligned}\texttt{obj} : [\, \forall\, \alpha_2\ \alpha_{46}\ \alpha_{57}\ .\ \alpha_2 \mid\ & \\ & \alpha_{46} \to \alpha_{57} \ \leqslant\ \alpha_2 \\ & \wedge\ \alpha_{46} \ \leqslant\ \{\, \texttt{I} \parallel \texttt{S} \,\} \\ & \wedge\ (\alpha_{46} \,\#\, \texttt{I} \ \vee\ \texttt{int} \leqslant \alpha_{57}) \\ & \wedge\ (\alpha_{46} \,\#\, \texttt{S} \ \vee\ \texttt{string} \leqslant \alpha_{57})\,]\end{aligned}$$

L'appel `f obj` est alors refusé par le typeur puisque la saturation des contraintes engendre en particulier les contraintes suivantes (dans lesquelles on s'abstrait des renommages de variables dues aux instanciations qui, en l'occurrence, ne changent pas le résultat) :

- $(\alpha_4 \leqslant \alpha_{46})$ et $(\alpha_{57} \leqslant \alpha_5)$ (dus à $(\alpha_{46} \to \alpha_{57} \leqslant \alpha_2)$ et $(\alpha_2 \leqslant \alpha_4 \to \alpha_5)$)
- $(\alpha_6 \leqslant \alpha_{46})$ et $(\alpha_{57} \leqslant \alpha_7)$ (dus à $(\alpha_{46} \to \alpha_{57} \leqslant \alpha_2)$ et $(\alpha_2 \leqslant \alpha_6 \to \alpha_7)$)
- $\texttt{S} \,\#\, \texttt{S} \vee \texttt{string} \leqslant \alpha_{57}$ (dû à $(\texttt{S} \leqslant \alpha_4)$, $(\alpha_4 \leqslant \alpha_{46})$ et $(\alpha_{46} \,\#\, \texttt{S} \vee \texttt{string} \leqslant \alpha_{57})$)
- $\texttt{I} \,\#\, \texttt{I} \vee \texttt{int} \leqslant \alpha_{57}$ (dû à $(\texttt{I} \leqslant \alpha_6)$, $(\alpha_6 \leqslant \alpha_{46})$ et $(\alpha_{46} \,\#\, \texttt{I} \vee \texttt{int} \leqslant \alpha_{57})$)
- $\texttt{string} \leqslant \alpha_{57}$ (dû à $(\texttt{S} \,\#\, \texttt{S} \vee \texttt{string} \leqslant \alpha_{57})$)
- $\texttt{int} \leqslant \alpha_{57}$ (dû à $(\texttt{I} \,\#\, \texttt{I} \vee \texttt{int} \leqslant \alpha_{57})$)
- $\texttt{string} \leqslant \texttt{int}$ (dû à $(\texttt{string} \leqslant \alpha_{57})$, $(\alpha_{57} \leqslant \alpha_7)$ et $(\alpha_7 \leqslant \texttt{int})$)
- $\texttt{int} \leqslant \texttt{string}$ (dû à $(\texttt{int} \leqslant \alpha_{57})$, $(\alpha_{57} \leqslant \alpha_5)$ et $(\alpha_5 \leqslant \texttt{string})$)

et donc de multiples clashs.

Néanmoins, ce code est valide (au sens où il s'exécutera sans erreur), et ce serait un gain notable pour l'expressivité du langage que le typeur l'accepte.

4.3 Principes généraux du nouveau système de types

L'idée de base consiste à généraliser le type des arguments des fonctions et à retarder le plus possible l'instanciation des schémas ainsi produits. Ces instanciations seront alors repoussées dans les sous-arbres de saturation.

Intuitivement, cela revient, sur l'exemple précédent, à re-généraliser le type de `obj` (venant d'être instancié) en un schéma de type σ lors de son passage à `f`, et à propager σ jusqu'à l'obtention des deux contraintes :

- $\sigma \leqslant \alpha_4 \to \alpha_5$
- $\sigma \leqslant \alpha_6 \to \alpha_7$

Dans cette situation, puisque le schéma est comparé à des types construits, la saturation des contraintes engendre deux instanciations indépendantes produisant les deux ensembles de contraintes suivants :

- $\alpha'_2 \leqslant \alpha_4 \to \alpha_5$
- $\alpha'_{46} \to \alpha'_{57} \leqslant \alpha'_2$
- $\alpha'_{46} \leqslant \{\, \mathtt{I} \parallel \mathtt{S} \,\}$
- $(\alpha'_{46} \,\#\, \mathtt{I} \vee \mathtt{int} \leqslant \alpha'_{57})$
- $(\alpha'_{46} \,\#\, \mathtt{S} \vee \mathtt{string} \leqslant \alpha'_{57})$
- $\alpha''_2 \leqslant \alpha_6 \to \alpha_7$
- $\alpha''_{46} \to \alpha''_{57} \leqslant \alpha''_2$
- $\alpha''_{46} \leqslant \{\, \mathtt{I} \parallel \mathtt{S} \,\}$
- $(\alpha''_{46} \,\#\, \mathtt{I} \vee \mathtt{int} \leqslant \alpha''_{57})$
- $(\alpha''_{46} \,\#\, \mathtt{S} \vee \mathtt{string} \leqslant \alpha''_{57})$

La saturation de toutes les contraintes provenant de ces deux instanciations et de l'instanciation du schéma de type associé à `f` est alors possible. En effet, nous obtenons maintenant les contraintes :

- $(\alpha_4 \leqslant \alpha'_{46})$ et $(\alpha'_{57} \leqslant \alpha_5)$ (dus à $(\alpha'_{46} \to \alpha'_{57} \leqslant \alpha'_2)$ et $(\alpha'_2 \leqslant \alpha_4 \to \alpha_5)$)
- $(\alpha_6 \leqslant \alpha''_{46})$ et $(\alpha''_{57} \leqslant \alpha_7)$ (dus à $(\alpha''_{46} \to \alpha''_{57} \leqslant \alpha''_2)$ et $(\alpha''_2 \leqslant \alpha_6 \to \alpha_7)$)
- $\mathtt{S} \,\#\, \mathtt{S} \vee \mathtt{string} \leqslant \alpha'_{57}$ (dû à $(\mathtt{S} \leqslant \alpha_4)$, $(\alpha_4 \leqslant \alpha'_{46})$ et $(\alpha'_{46} \,\#\, \mathtt{S} \vee \mathtt{string} \leqslant \alpha'_{57})$)
- $\mathtt{I} \,\#\, \mathtt{I} \vee \mathtt{int} \leqslant \alpha''_{57}$ (dû à $(\mathtt{I} \leqslant \alpha_6)$, $(\alpha_6 \leqslant \alpha''_{46})$ et $(\alpha''_{46} \,\#\, \mathtt{I} \vee \mathtt{int} \leqslant \alpha''_{57})$)
- $\mathtt{string} \leqslant \alpha'_{57}$ (dû à $(\mathtt{S} \,\#\, \mathtt{S} \vee \mathtt{string} \leqslant \alpha'_{57})$)
- $\mathtt{int} \leqslant \alpha''_{57}$ (dû à $(\mathtt{I} \,\#\, \mathtt{I} \vee \mathtt{int} \leqslant \alpha''_{57})$)
- $\mathtt{string} \leqslant \mathtt{string}$ (dû à $(\mathtt{string} \leqslant \alpha'_{57})$, $(\alpha'_{57} \leqslant \alpha_5)$ et $(\alpha_5 \leqslant \mathtt{string})$)
- $\mathtt{int} \leqslant \mathtt{int}$ (dû à $(\mathtt{int} \leqslant \alpha''_{57})$, $(\alpha''_{57} \leqslant \alpha_7)$ et $(\alpha_7 \leqslant \mathtt{int})$)

dans lesquelles `int` et `string` ne provoquent plus de clash.

Un problème se pose cependant : puisque la saturation va maintenant instancier des schémas de type, elle va générer de nouvelles variables de type. Or, la terminaison de la saturation était jusqu'ici assurée par le fait que les types sont en nombre fini dans les sous-arbres de saturation. Créer de nouvelles variables de type de manière incontrôlée lors de la construction des sous-arbres de saturation remet en cause la preuve de terminaison, et ce pour de bonnes raisons puisqu'un tel typeur bouclerait effectivement sur certains programmes.

Pour résoudre ce problème, nous allons introduire une technique permettant de limiter le nombre de variables de type générées dans les sous-arbres de saturation. Une telle limitation ne doit cependant pas être construite de manière complètement arbitraire. En effet, on souhaite conserver une certaine « logique » compréhensible par le programmeur, et être capable d'expliquer pourquoi un programme n'est pas typable autrement que par « *eh bien voilà, un compteur a quelque part atteint une borne et on s'est arrêté de générer de nouvelles variables de type pendant la saturation des contraintes* ».

L'idée consiste à annoter les relations de sous-typage (c'est-à-dire les $(\leqslant)$) avec une clé. Ces clés sont simplement une séquence d'entiers, chaque entier correspondant à des numéros d'occurrence de paramètres dans des corps de fonctions. Dans la phase de saturation, chaque fois qu'une instanciation doit être réalisée suite à l'obtention d'une relation de la forme $(\sigma \leqslant_k \tau^r)$ avec τ^r un type construit, nous utilisons la clé k pour instancier le schéma σ de manière déterministe en fonction de k.

Plus précisément, nous définissons la grammaire des clés k ainsi :

η `::= 0 | 1 | 2 | ...`

k `::=` $[\eta_1, \eta_2, ..., \eta_p]$ (avec $p \geqslant 0$, une clé pouvant être vide)

Nous nous munissons d'un opérateur, noté $(\times)$, prenant en argument une clé k et une variable de type α et générant une variable de type α' différente de toutes les autres variables de type qui n'ont pas été générées par la même opération $(k \times \alpha)$ entre ce k donné et ce α donné. Mathématiquement parlant, l'opérateur $(\times)$ est simplement une fonction injective.

Un tel opérateur nous permet de définir une technique d'« instanciation paramétrée par une clé » qui, à partir d'un schéma de la forme $[\ \forall\alpha_1...\alpha_n\ .\ \alpha_0 \mid \Phi_0\ ]$ et d'une clé k calcule les n variables de type suivantes :

- $\alpha'_1 = k \times \alpha_1$
- $\alpha'_2 = k \times \alpha_2$
- ...
- $\alpha'_n = k \times \alpha_n$

et les utilise pour remplacer $\alpha_1, ..., \alpha_n$ en calculant $\alpha_0[\alpha_i \mapsto \alpha'_i]_{i=1}^n$ et $\Phi_0[\alpha_i \mapsto \alpha'_i]_{i=1}^n$ comme lors d'une instanciation classique. La seule différence avec une instanciation classique est donc due au fait que les α'_i ne sont pas « frais » mais calculés à partir de k et des α_i originaux.

Dans le contexte de ce nouveau système, la construction de langage `(let x =` e_1 `in` e_2`)` n'a plus lieu d'être. Elle est alors simplement considérée comme du sucre syntaxique sur `((`λ `x . `e_2`)` e_1`)`. En conséquence, plus aucune règle d'inférence n'ajoute de schéma de type dans l'environnement de typage Γ. Nous pouvons alors simplifier la définition de Γ en :

Γ `::= (`x_1`,` α_1`), ..., (`x_n`,` α_n`)`

Il reste à définir comment calculer les clés d'instanciation pour qu'elles soient « différentes » lorsqu'il est intéressant qu'un même schéma de type soit instancié avec des variables de type différentes. L'astuce consiste à annoter les variables, constantes et primitives du programme par un numéro représentant leur occurrence dans le code. Plusieurs formalismes sont possibles pour encoder une telle opération lors du typage. Une possibilité serait de transporter à travers les règles d'inférence un ensemble Ξ de couples de la forme $((\mathtt{x} \mid \mathtt{c} \mid \mathtt{p}^1 \mid \mathtt{p}^2),\ \eta)$ représentant les indices

courants des différentes variables vivantes, constantes et primitives du programme. Les règles de typage seraient alors de la forme :

$$\frac{\mathrm{T}\ldots \quad \ldots \quad \ldots \quad \ldots}{\Xi, \Phi, \Gamma \vdash \mathtt{e} : \alpha \rhd \Xi', \Phi'}$$

Ce formalisme, bien qu'assez élégant, apporterait une certaine lourdeur à tout le système de types, et nous préférons ici supposer simplement qu'une première passe sur le code a annoté les variables, constantes et primitives du programme avec leurs indices d'occurrence respectifs. Nous remarquerons que l'annotation des constantes et primitives n'a de réel intérêt que pour celles qui sont polymorphes comme par exemple la liste vide `[]` et `fst`. De plus, l'annotation des variables n'est utile que pour celles apparaissant plusieurs fois. Il est donc possible par ce biais de limiter le nombre d'annotations et ainsi la longueur des clés, chose qui se révèlera utile par la suite. Nous étendons alors les expressions avec la construction :

$$\mathtt{e} \mathrel{+}= \mathtt{x}^{\eta} \mid \mathtt{c}^{\eta} \mid \mathtt{p}^{1,\eta}\ \mathtt{e} \mid \mathtt{p}^{2,\eta}\ \mathtt{e}_1\ \mathtt{e}_2$$

où η représente l'indice d'occurrence de la variable, constante ou primitive dans l'expression. Cet indice, η, est alors utilisé dans les règles de typage des variables, constantes et applications de primitives pour annoter les relations de sous-typage qu'elles génèrent. Ces clés sont ensuite transportées des conclusions aux prémisses lors de la construction de l'arbre d'inférence. Enfin, lors de l'application de la transitivité de $(\leqslant)$ transformant plusieurs contraintes de sous-typage en une, typiquement lorsque l'algorithme de saturation aboutit à deux contraintes de la forme $(\tau^l \leqslant_{k_1} \alpha)$ et $(\alpha \leqslant_{k_2} \tau^r)$, la contrainte générée est $(\tau^l \leqslant_{k_1 @ k_2} \tau^r)$ où $(@)$ désigne la concaténation des clés.

Pour assurer la terminaison de l'algorithme d'inférence, nous devons maintenant trouver un moyen de borner le nombre de clés d'une manière qui ait du sens vis-à-vis de la sémantique du programme. Pour ce faire, nous les tronquons simplement à une certaine longueur. Cette longueur limite est un paramètre du système de types, et la valeur de ce paramètre définit le « niveau de polymorphisme » du typeur correspondant. Nous verrons plus tard (dans la section 4.7) quelle « signification » donner à cette limite, et en particulier comment « interpréter » le niveau de polymorphisme minimal nécessaire à l'acceptation d'un programme.

4.4 Nouvelles règles d'inférence

Définissons maintenant les règles d'inférence correspondant à ce nouveau système. Pour simplifier, nous travaillons dans le reste de ce chapitre sur une version épurée du langage, en particulier sans la construction $(\mathtt{let\ x = e_1\ in\ e_2})$, maintenant vue comme du sucre syntaxique, mais aussi sans variants polymorphes ni filtrages de motifs. Les variants n'ont en effet aucun lien avec l'extension que nous présentons ici. Leur ajout est trivial mais provoque une augmentation inutile du nombre de règles.

Les types que nous manipulons sont maintenant définis par les grammaires suivantes :

τ^l ::= α | (α_1, ..., α_n) t | σ

τ^r ::= α | (α_1, ..., α_n) t

Nous remarquerons en particulier la présence de σ dans la définition de τ^l mais pas dans la définition de τ^r. En effet, de par la définition des règles d'inférence que nous allons donner, les schémas de type seront toujours générés à gauche d'un ($\leqslant$), et il serait difficile de donner un sens à leur présence à droite d'un ($\leqslant$).

Par ailleurs, les environnements de typage sont, comme nous l'avons dit précédemment, définis par la grammaire :

Γ ::= (x_1, α_1), ..., (x_n, α_n)

Pour les mêmes raisons de simplicité, le système que nous définissons ici ne gère pas non plus l'extension du filtrage du chapitre précédent. Les ensembles de contraintes que nous manipulons ne contiennent donc plus de disjonctions comme c'était le cas dans le système précédent et sont définis par la grammaire Φ comme suit :

η ::= 0 | 1 | 2 | ...

k ::= $[\eta_1, \eta_2, ..., \eta_p]$

C ::= $\tau^l \leqslant_k \tau^r$

Φ ::= $C_1 \wedge ... \wedge C_n$

très similaire à celle du chapitre 2. La seule différence notable tient à la présence de la clé k en indice des relations de sous-typage ($\leqslant$).

Par ailleurs, nos règles d'inférence sont toujours classifiables dans les trois catégories habituelles :

- Les **règles de typage** sont de la même forme que pour le système de base :

$$\frac{\text{T}\ldots \quad \ldots \quad \ldots \quad \ldots}{\Phi, \Gamma \vdash \mathtt{e} : \alpha \rhd \Phi'}$$

- La **règle d'instanciation** est maintenant de la forme :

$$\frac{\text{I}\ldots \quad \ldots \quad \ldots \quad \ldots}{\Phi \vdash \sigma \leq_k (\alpha_1, \ldots, \alpha_n)\ t \rhd \Phi'}$$

  De manière générale, la règle d'instanciation s'applique sur toute comparaison entre un schéma de type et un type construit. La forme « (α_1, ..., α_n) t » (recouvrant comme avant l'ensemble des types de base, le produit cartésien ($\times$) et la flèche ($\rightarrow$)) est ici la seule correspondant à un type construit dans τ_r. En cas d'ajout de variants polymorphes ou d'autres catégories de types construits, il faudrait étendre cette règle d'instanciation en conséquence.

- Les **règles de saturation** sont maintenant de l'une des formes :

$$\frac{\text{S}\ldots \quad \ldots \quad \ldots \quad \ldots}{\Phi \vdash \alpha \leqslant_k \tau^r \rhd \Phi'} \qquad \frac{\text{S}\ldots \quad \ldots \quad \ldots \quad \ldots}{\Phi \vdash (\alpha_1, \ldots, \alpha_n)\ t \leqslant_k \tau^r \rhd \Phi'}$$

$$\frac{\text{S}\ldots \quad \ldots \quad \ldots \quad \ldots}{\Phi \vdash \alpha \leq_k \tau^r \rhd \Phi'} \qquad \frac{\text{S}\ldots \quad \ldots \quad \ldots \quad \ldots}{\Phi \vdash (\alpha_1, \ldots, \alpha_n)\ t \leq_k \tau_r \rhd \Phi'}$$

  Nous noterons la présence de k en indice des comparaisons ($\leqslant$) comme dans la règle d'instanciation.

Le mécanisme de généralisation/instanciation ayant complètement changé, l'utilisation de ces règles présente néanmoins une différence notable avec les systèmes précédents : les règles d'instanciation ne sont maintenant plus appliquées immédiatement après certaines règles de typage mais au milieu des sous-arbres de saturation.

Nouvelles règles de typage

Définissons maintenant les règles de typage de notre nouveau système.

- Le typage des constantes est défini par les deux règles suivantes :

$$\text{TConst}\ \frac{\Phi \vdash T(\mathtt{c}) \leqslant_{[]} \alpha \rhd \Phi'}{\Phi, \Gamma \vdash \mathtt{c} : \alpha \rhd \Phi'} \qquad \text{TConstPoly}\ \frac{\Phi \vdash T(\mathtt{c}) \leqslant_{[\eta]} \alpha \rhd \Phi'}{\Phi, \Gamma \vdash \mathtt{c}^{\eta} : \alpha \rhd \Phi'}$$

  Comme habituellement, nous utilisons la fonction T de typage des constantes (et des primitives) renvoyant un schéma de type. Pour les constantes polymorphes (comme par exemple la liste vide), leur indice d'occurrence η est utilisé pour construire la clé $[\eta]$ indiçant la contrainte de sous-typage générée.

  Le fait de distinguer les constantes polymorphes des constantes non-polymorphes est facultatif. Il est tout à fait valide d'annoter toutes les constantes, tout comme de n'en annoter aucune. Cela aurait pour unique conséquence une perte de polymorphisme.

- Le typage des variables est similaire :

$$\text{TVar}\ \frac{\texttt{when}\ \Gamma[\mathtt{x}]\ \textit{defined} \qquad \Phi \vdash \Gamma[\mathtt{x}] \leqslant_{[]} \alpha \rhd \Phi'}{\Phi, \Gamma \vdash \mathtt{x} : \alpha \rhd \Phi'} \qquad \text{TVarPoly}\ \frac{\texttt{when}\ \Gamma[\mathtt{x}]\ \textit{defined} \qquad \Phi \vdash \Gamma[\mathtt{x}] \leqslant_{[\eta]} \alpha \rhd \Phi'}{\Phi, \Gamma \vdash \mathtt{x}^{\eta} : \alpha \rhd \Phi'}$$

  à la nuance près que $\Gamma[\mathtt{x}]$ est maintenant une variable de type. Cette règle ne donne donc plus immédiatement lieu à l'application d'une règle d'instanciation.

- Le typage des couples est quant à lui très simple :

$$\textsc{TPair}\quad \frac{\texttt{let}\ \alpha_1, \alpha_2\ \textit{fresh} \qquad \Phi, \Gamma \vdash \texttt{e}_1 : \alpha_1 \rhd \Phi' \qquad \Phi', \Gamma \vdash \texttt{e}_2 : \alpha_2 \rhd \Phi'' \qquad \Phi'' \vdash \alpha_1 \times \alpha_2 \leqslant_{[]} \alpha \rhd \Phi'''}{\Phi, \Gamma \vdash (\texttt{e}_1,\ \texttt{e}_2) : \alpha \rhd \Phi'''}$$

  La seule contrainte de sous-typage qu'il génère, à savoir $(\alpha_1 \times \alpha_2 \leqslant_{[]} \alpha)$ fait intervenir une clé vide.

- Le typage de la conditionnelle est lui aussi trivial :

$$\textsc{TIf}\quad \frac{\texttt{let}\ \alpha'\ \textit{fresh} \qquad \Phi \vdash \alpha' \leqslant_{[]} \texttt{bool} \rhd \Phi' \qquad \Phi', \Gamma \vdash \texttt{e}_1 : \alpha' \rhd \Phi'' \qquad \Phi'', \Gamma \vdash \texttt{e}_2 : \alpha \rhd \Phi''' \qquad \Phi''', \Gamma \vdash \texttt{e}_3 : \alpha \rhd \Phi''''}{\Phi, \Gamma \vdash \texttt{if}\ \texttt{e}_1\ \texttt{then}\ \texttt{e}_2\ \texttt{else}\ \texttt{e}_3 : \alpha \rhd \Phi''''}$$

  Aucun polymorphisme n'est nécessaire vis-à-vis du booléen correspondant au test. La clé reliant α' et `bool` est donc vide.

- La règle de typage du λ est presque identique à celle des systèmes précédents :

$$\textsc{TLambda}\quad \frac{\texttt{let}\ \alpha_1, \alpha_2\ \textit{fresh} \qquad \Phi, \Gamma \oplus (\texttt{x}, \alpha_1) \vdash \texttt{e} : \alpha_2 \rhd \Phi' \qquad \Phi' \vdash \alpha_1 \to \alpha_2 \leqslant_{[]} \alpha \rhd \Phi''}{\Phi, \Gamma \vdash \lambda\ \texttt{x}\ .\ \texttt{e} : \alpha \rhd \Phi''}$$

- Les règles intéressantes vis-à-vis du polymorphisme sont celles correspondant au typage d'une application, et en particulier la règle suivante :

$$\textsc{TApp}\quad \frac{\texttt{let}\ \alpha_1, \alpha_2, \alpha_3, \alpha_4\ \textit{fresh} \qquad \Phi_0 \vdash \alpha_1 \leqslant_{[]} \alpha_3 \to \alpha_4 \rhd \Phi_1 \qquad \Phi_1 \vdash \alpha_4 \leqslant_{[]} \alpha \rhd \Phi_2 \qquad \Phi_2, \Gamma \vdash \texttt{e}_1 : \alpha_1 \rhd \Phi_3 \qquad \Phi_3, \Gamma \vdash \texttt{e}_2 : \alpha_2 \rhd \Phi_4 \qquad \Phi_4 \vdash \textsc{gen}(\alpha_2, \Phi_4, \Gamma) \leqslant_{[]} \alpha_3 \rhd \Phi_5}{\Phi_0, \Gamma \vdash \texttt{e}_1\ \texttt{e}_2 : \alpha \rhd \Phi_5}$$

  Nous commençons par lier la variable de type α_1 avec le type fonctionnel $\alpha_3 \to \alpha_4$ en générant la contrainte $(\alpha_1 \leqslant_{[]} \alpha_3 \to \alpha_4)$. Nous relions ensuite α_4 et α par la contrainte de sous-typage $(\alpha_4 \leqslant_{[]} \alpha)$. Nous typons les expressions $\texttt{e}_1$ et $\texttt{e}_2$ avec respectivement les variables de type α_1 et α_2. Nous généralisons enfin α_2 avec l'ensemble de contraintes Φ_4 généré. Comme habituellement, les variables de type apparaissant libres dans Γ ne sont pas généralisées (sachant que, d'après la structure de Γ, toutes les variables y apparaissant sont libres).

  Le chaînage des ensembles de contraintes défini par cette règle a été choisi de telle sorte que l'ordre de typage suive l'ordre du code. Il est en réalité possible d'effectuer différentes permutations sans changer l'ensemble des expressions acceptées par le système de types, la seule contrainte est d'effectuer la généralisation de α_2 après le typage de $\texttt{e}_2$. Pour limiter la taille de l'ensemble de contraintes à analyser lors de la généralisation, il pourrait en

particulier être intéressant de commencer par le typage de $\mathtt{e}_2$.

Le typage de l'application d'une primitive fait intervenir le même mécanisme. Il est défini par les quatre règles suivantes, le choix de la règle se faisant en fonction de l'arité de la primitive et du fait qu'elle est polymorphe ou non (comme pour les constantes). Ces règles sont en réalité un mélange des règles TApp et TConst. Elles n'apportent rien au discours actuel, nous ne les donnons que par souci d'exhaustivité :

TApplyPrim1

$$\frac{\begin{array}{c}\mathtt{let}\ \alpha_1,\alpha_2,\alpha_3\ \textit{fresh} \qquad \Phi_0 \vdash T(\mathtt{p}^1) \leqslant_{[]} \alpha_2 \to \alpha_3 \rhd \Phi_1 \\ \Phi_1 \vdash \alpha_3 \leqslant_{[]} \alpha \rhd \Phi_2 \qquad \Phi_2, \Gamma \vdash \mathtt{e}_1 : \alpha_1 \rhd \Phi_3 \qquad \Phi_3 \vdash \textsc{gen}(\alpha_1, \Phi_3, \Gamma) \leqslant_{[]} \alpha_2 \rhd \Phi_4\end{array}}{\Phi_0, \Gamma \vdash \mathtt{p}^1\ \mathtt{e}_1 : \alpha \rhd \Phi_4}$$

TApplyPrim1Poly

$$\frac{\begin{array}{c}\mathtt{let}\ \alpha_1,\alpha_2,\alpha_3\ \textit{fresh} \qquad \Phi_0 \vdash T(\mathtt{p}^1) \leqslant_{[\eta]} \alpha_2 \to \alpha_3 \rhd \Phi_1 \\ \Phi_1 \vdash \alpha_3 \leqslant_{[\eta]} \alpha \rhd \Phi_2 \qquad \Phi_2, \Gamma \vdash \mathtt{e}_1 : \alpha_1 \rhd \Phi_3 \qquad \Phi_3 \vdash \textsc{gen}(\alpha_1, \Phi_3, \Gamma) \leqslant_{[]} \alpha_2 \rhd \Phi_4\end{array}}{\Phi_0, \Gamma \vdash \mathtt{p}^{1,\eta}\ \mathtt{e}_1 : \alpha \rhd \Phi_4}$$

TApplyPrim2

$$\frac{\begin{array}{c}\mathtt{let}\ \alpha_0,\alpha_1,\alpha_2,\alpha_3,\alpha_4,\alpha_5\ \textit{fresh} \\ \Phi_0 \vdash T(\mathtt{p}^2) \leqslant_{[]} \alpha_2 \to \alpha_0 \rhd \Phi_1 \qquad \Phi_1 \vdash \alpha_0 \leqslant_{[]} \alpha_4 \to \alpha_5 \rhd \Phi_2 \\ \Phi_2 \vdash \alpha_5 \leqslant_{[]} \alpha \rhd \Phi_3 \qquad \Phi_3, \Gamma \vdash \mathtt{e}_1 : \alpha_1 \rhd \Phi_4 \qquad \Phi_4 \vdash \textsc{gen}(\alpha_1, \Phi_4, \Gamma) \leqslant_{[]} \alpha_2 \rhd \Phi_5 \\ \Phi_5, \Gamma \vdash \mathtt{e}_2 : \alpha_3 \rhd \Phi_6 \qquad \Phi_6 \vdash \textsc{gen}(\alpha_3, \Phi_6, \Gamma) \leqslant_{[]} \alpha_4 \rhd \Phi_7\end{array}}{\Phi_0, \Gamma \vdash \mathtt{p}^2\ \mathtt{e}_1\ \mathtt{e}_2 : \alpha \rhd \Phi_7}$$

TApplyPrim2Poly

$$\frac{\begin{array}{c}\mathtt{let}\ \alpha_0,\alpha_1,\alpha_2,\alpha_3,\alpha_4,\alpha_5\ \textit{fresh} \\ \Phi_0 \vdash T(\mathtt{p}^2) \leqslant_{[\eta]} \alpha_2 \to \alpha_0 \rhd \Phi_1 \qquad \Phi_1 \vdash \alpha_0 \leqslant_{[]} \alpha_4 \to \alpha_5 \rhd \Phi_2 \\ \Phi_2 \vdash \alpha_5 \leqslant_{[]} \alpha \rhd \Phi_3 \qquad \Phi_3, \Gamma \vdash \mathtt{e}_1 : \alpha_1 \rhd \Phi_4 \qquad \Phi_4 \vdash \textsc{gen}(\alpha_1, \Phi_4, \Gamma) \leqslant_{[]} \alpha_2 \rhd \Phi_5 \\ \Phi_5, \Gamma \vdash \mathtt{e}_2 : \alpha_3 \rhd \Phi_6 \qquad \Phi_6 \vdash \textsc{gen}(\alpha_3, \Phi_6, \Gamma) \leqslant_{[]} \alpha_4 \rhd \Phi_7\end{array}}{\Phi_0, \Gamma \vdash \mathtt{p}^{2,\eta}\ \mathtt{e}_1\ \mathtt{e}_2 : \alpha \rhd \Phi_7}$$

Nouvelle règle d'instanciation

La règle d'instanciation, au lieu de générer des variables de type fraîches, utilise maintenant l'opérateur $(\times)$ pour générer des variables de type en utilisant la clé qui a été agrégée :

Inst

$$\frac{\begin{array}{c}\mathtt{let}\ \alpha'_1,\ldots,\alpha'_n = k \times \alpha_1,\ \ldots,\ k \times \alpha_n \\ \Phi \vdash \alpha_0[\alpha_i \mapsto \alpha'_i]_{i=1}^n \leqslant_k \tau^r \rhd \Phi_1 \qquad \Phi_1 \vdash \mathtt{C}_1[\alpha_i \mapsto \alpha'_i]_{i=1}^n \rhd \Phi_2 \quad \cdots \quad \Phi_p \vdash \mathtt{C}_p[\alpha_i \mapsto \alpha'_i]_{i=1}^n \rhd \Phi_{p+1}\end{array}}{\Phi \vdash [\ \forall \alpha_1\ \ldots\ \alpha_n\ .\ \alpha_0 \mid \mathtt{C}_1 \wedge \cdots \wedge \mathtt{C}_p\ ] \leq_k \tau^r \rhd \Phi_{p+1}}$$

Comme nous l'avons dit précédemment, cette règle n'est plus appliquée au dessus d'une règle de typage, mais dans les sous-arbres de saturation. Nous remarquerons que sa conclusion mentionne

un ($\leq$) et pas un ($\leqslant$) comme avant. Cela lui impose en particulier d'être appliquée au dessus de la règle de saturation SNewConstraint.

Nouvelles règles de saturation

Les règles de saturation sont très similaires à celles du système de base. La principale différence est qu'elles propagent maintenant les clés indiçant les relations de sous-typage :

- Les règles permettant le calcul de point fixe sont les suivantes :

$$\textsc{SNewConstraint}\ \frac{\texttt{when}\ (\tau^l \leqslant_k \tau^r) \notin \Phi \qquad \Phi \wedge \tau^l \leqslant_k \tau^r \vdash \tau^l \leq_k \tau^r \rhd \Phi'}{\Phi \vdash \tau^l \leqslant_k \tau^r \rhd \Phi'} \qquad \textsc{SAlreadyProved}\ \frac{\texttt{when}\ (\tau^l \leqslant_k \tau^r) \in \Phi}{\Phi \vdash \tau^l \leqslant_k \tau^r \rhd \Phi}$$

- Comme dans les systèmes précédents, la gestion d'une relation de sous-typage entre deux types construits de même nom se fait en liant les arguments des types construits deux à deux. Ce comportement est décrit par la règle suivante :

$$\textsc{STypeConstr}\ \frac{\begin{array}{c}\Phi_1 \vdash \alpha_1 \leqslant_k \alpha'_1 \rhd \Phi'_1 \qquad \Phi'_1 \vdash \alpha'_1 \leqslant_k \alpha_1 \rhd \Phi_2 \\ \cdots \\ \Phi_n \vdash \alpha_n \leqslant_k \alpha'_n \rhd \Phi'_n \qquad \Phi'_n \vdash \alpha'_n \leqslant_k \alpha_n \rhd \Phi_{n+1}\end{array}}{\Phi_1 \vdash (\alpha_1, \ldots, \alpha_n)\ t \leq_k (\alpha'_1, \ldots, \alpha'_n)\ t \rhd \Phi_{n+1}}$$

  La clé est simplement propagée comme pour les autres règles de saturation.

- La gestion des relations de sous-typage liant des variables de type se fait par les règles suivantes :
  - Deux variables de type de même nom sont toujours compatibles entre elles :

$$\textsc{SSameVar}\ \frac{}{\Phi \vdash \alpha \leq_k \alpha \rhd \Phi}$$

  - Pour la gestion de la transitivité, nous étendons les définitions des fonctions lefts et rights de manière à extraire de Φ, en plus des types, les clés qui ont permis de lier ces types avec la variable recherchée :

$$\textsc{lefts}(\alpha,\ \Phi) \triangleq \{\ (k, \tau^l) \mid (\tau^l \leqslant_k \alpha) \in \Phi\ \}$$
$$\textsc{rights}(\alpha,\ \Phi) \triangleq \{\ (k, \tau^r) \mid (\alpha \leqslant_k \tau^r) \in \Phi\ \}$$

    Grâce à ces deux fonctions, les trois règles habituelles de gestion de la transitivité peuvent

s'écrire de la manière suivante :

$$\textsc{STransRight}\quad \frac{\texttt{when}\ \tau^l \notin \{\,\alpha\,\} \qquad \texttt{let}\ (k_1, \tau_1^r), \ldots, (k_n, \tau_n^r) = \textsc{rights}(\alpha,\ \Phi_1) \qquad \Phi_1 \vdash \tau^l \leqslant_{k@k_1} \tau_1^r \rhd \Phi_2 \qquad \cdots \qquad \Phi_n \vdash \tau^l \leqslant_{k@k_n} \tau_n^r \rhd \Phi_{n+1}}{\Phi_1 \vdash \tau^l \leq_k \alpha \rhd \Phi_{n+1}}$$

$$\textsc{STransLeft}\quad \frac{\texttt{when}\ \tau^r \notin \{\,\alpha\,\} \qquad \texttt{let}\ (k_1, \tau_1^l), \ldots, (k_n, \tau_n^l) = \textsc{lefts}(\alpha,\ \Phi_1) \qquad \Phi_1 \vdash \tau_1^l \leqslant_{k_1@k} \tau^r \rhd \Phi_2 \qquad \cdots \qquad \Phi_n \vdash \tau_n^l \leqslant_{k_n@k} \tau^r \rhd \Phi_{n+1}}{\Phi_1 \vdash \alpha \leq_k \tau^r \rhd \Phi_{n+1}}$$

$$\textsc{STransLeftRight}\quad \frac{\begin{array}{c}\texttt{when}\ \alpha_1 \neq \alpha_2 \\ \texttt{let}\ (k_1, \tau_1^l), \ldots, (k_n, \tau_n^l) = \textsc{lefts}(\alpha_1,\ \Phi_{1,1}) \qquad \texttt{let}\ (k'_1, \tau_1^r), \ldots, (k'_p, \tau_p^r) = \textsc{rights}(\alpha_2,\ \Phi_{1,1}) \\ \Phi_{1,1} \vdash \tau_1^l \leqslant_{k_1@k@k'_1} \tau_1^r \rhd \Phi_{1,2} \qquad \cdots \qquad \Phi_{1,p} \vdash \tau_1^l \leqslant_{k_1@k@k'_p} \tau_p^r \rhd \Phi_{2,1} \\ \vdots \qquad\qquad\qquad\qquad \vdots \\ \Phi_{n,1} \vdash \tau_n^l \leqslant_{k_n@k@k'_1} \tau_1^r \rhd \Phi_{n,2} \qquad \cdots \qquad \Phi_{n,p} \vdash \tau_n^l \leqslant_{k_n@k@k'_p} \tau_p^r \rhd \Phi_{n+1,1}\end{array}}{\Phi_{1,1} \vdash \alpha_1 \leq_k \alpha_2 \rhd \Phi_{n+1,1}}$$

Dans ces règles, nous avons pris garde à concaténer les clés dans le même sens que les relations dont elles proviennent. Bien que ce choix n'impacte pas la validité du système, il permet de maintenir une certaine stabilité des clés par rapport à l'ordre d'arrivée des contraintes de sous-typage dans Φ. Nous évitons alors qu'une même contrainte générée par deux biais différents possède une permutation différente de sa clé et ainsi limitons la croissance de Φ.

4.5 Terminaison de l'algorithme d'inférence

Comme nous l'avons dit, pour assurer la terminaison, nous bornons la longueur des clés. Nous montrons ici que cette propriété est effectivement suffisante. La seule difficulté consiste à montrer que le calcul de point fixe appliqué dans les sous-arbres de saturation se termine.

Tout d'abord, nous remarquons que les indices d'occurrence de variables, constantes et primitives sont en nombre fini. En effet, dans le pire cas, il y en a autant qu'il n'y a de noeuds dans l'AST du programme. Les clés, constituées de séquences de taille bornée d'indices sont donc également en nombre fini.

Par ailleurs, les instances de règles de typage sont en nombre fini pour les raisons habituelles. Seules des règles de typage génèrent de nouveaux schémas de type (il s'agit des règles gérant une application, à savoir TApp, TApplyPrim1, TApplyPrim2, TApplyPrim1Poly et TApplyPrim2Poly). En particulier, aucune règle de saturation ne génère de nouveaux schémas de type, ces derniers sont donc en nombre fini dans tous les sous-arbres de saturation.

D'autre part, seule la règle Inst est susceptible de générer de nouvelles variables de type dans les sous-arbres de saturation en appliquant l'opérateur $\times$ sur une clé et une variable de type. Les variables de type données en deuxième argument de $(\times)$ proviennent systématiquement d'un schéma de type. Les schémas de type étant en nombre fini dans les sous-arbres de saturation, et les clés étant également en nombre fini, les nouvelles variables de type générées par des produits de la forme $k \times \alpha$ sont donc également en nombre fini. Nous avons donc montré que les variables de type manipulées dans les sous-arbres de saturation sont en nombre fini.

Nous en concluons que les types apparaissant dans chaque sous-arbre de saturation sont en nombre fini, et donc les contraintes de type également, ce qui implique que les ensembles de contraintes sont de taille bornée et donc que le calcul de point fixe appliqué sur l'ensemble de contraintes dans les sous-arbres de saturation termine.

4.6 Adaptation de la preuve de validité

Les clés sont utilisées lors de la génération de variables de type au moment de l'instanciation d'un schéma et servent uniquement à faire terminer l'algorithme d'inférence sans affecter la validité du système. Pour simplifier cette preuve, nous ignorerons simplement les clés et ne feront aucune supposition sur les variables de type générées lors d'instanciations, celles-ci pouvant être identiques ou différentes de celles générées lors de chaque instanciation.

La présence de schémas de type dans les contraintes de sous-typage complexifie grandement la preuve de validité de ce nouveau système. En effet, les relations d'ordre entre schémas de type doivent être adaptées et les lemmes de « monotonie » ne pourront plus s'exprimer de la même manière.

Pour bien comprendre l'origine de la définition de la relation d'ordre que nous allons donner, commençons par analyser l'évolution du schéma de type généré par notre mécanisme d'inférence sur les expressions obtenues à chaque étape de l'évaluation. Habituellement, cette évolution se résume à :

- la perte de certaines contraintes de sous-typage, typiquement due à l'élimination de certaines sous-expressions (comme par exemple lors de la réduction de `(if true then` $\mathtt{e}_1$ `else` $\mathtt{e}_2$`)` en $\mathtt{e}_1$)
- la réplication de certains sous-ensembles de contraintes, typiquement due à la réplication d'une valeur (comme par exemple lors de la réduction d'une expression de la forme `(let x =` $\mathtt{e}_1$ `in` $\mathtt{e}_2$`)` en $\mathtt{e}_2[\mathtt{x} \mapsto \mathtt{e}_1]$ lorsque `x` apparaît plusieurs fois libre dans $\mathtt{e}_2$).

Dans le contexte de ce nouveau système de types, le schéma de type inféré est susceptible de subir des transformations plus profondes. Considérons par exemple le cas de l'expression `((`λ `x . x) ((`λ `y . y) 42))` pour laquelle le schéma de type inféré est :

$$[\ \forall\alpha_1\ .\ \alpha_1 \mid [\ \forall\alpha_2\ .\ \alpha_2 \mid [\ \forall\alpha_3\ .\ \alpha_3 \mid \mathtt{int} \leqslant \alpha_3\ ] \leqslant \alpha_2\ ] \leqslant \alpha_1\ ]$$

Après une étape de réduction, nous obtenons l'expression `((λ x . x) 42)` pour laquelle le schéma de type inféré est :

$$[\ \forall\alpha_1\ .\ \alpha_1\ |\ [\ \forall\alpha_2\ .\ \alpha_2\ |\ \texttt{int} \leqslant \alpha_2\ ] \leqslant \alpha_1\ ]$$

Et après une troisième et dernière étape de réduction, nous obtenons l'expression `42` pour laquelle le schéma de type inféré est :

$$[\ \forall\alpha\ .\ \alpha\ |\ \texttt{int} \leqslant \alpha\ ]$$

Les lemmes de réduction de l'objet spécifient que le schéma de type inféré décroît au cours de l'évaluation. La relation d'ordre que nous devons définir entre les schémas de type doit alors prendre en compte le genre de transformation observée ci-dessus. Cette définition doit en particulier autoriser certaines relations de la forme $\sigma \leqslant \tau^r$ à être « éclatées » en faisant remonter les contraintes présentes dans σ au niveau du schéma englobant et en les reliant à τ^r. Une telle transformation revient à faire « remonter des $\forall$ dans les formules » comme c'est le cas lors de la mise en forme prénexe d'une formule logique. Il est possible de l'implémenter en une fonction de « normalisation » des schémas, ce qui permet de faciliter le nettoyage des contraintes et ainsi de limiter le nombre de variables de type générées lors des instanciations. L'autre avantage d'une telle normalisation est de simplifier l'affichage des schémas à l'utilisateur. Il sera typiquement plus pratique de lire :

```
((λ x . x) ((λ y . y) 42)) : int
```

que

$$\texttt{((λ x . x) ((λ y . y) 42)) :}\ [\ \forall\alpha_1\ .\ \alpha_1\ |\ [\ \forall\alpha_2\ .\ \alpha_2\ |\ [\ \forall\alpha_3\ .\ \alpha_3\ |\ \texttt{int} \leqslant \alpha_3\ ] \leqslant \alpha_2\ ] \leqslant \alpha_1\ ]$$

Néanmoins, il est important de remarquer qu'il ne faut pas éclater systématiquement toutes les contraintes de type de la forme $\sigma \leqslant \tau^r$, en particulier dans un schéma de type qui n'est pas clos et dans un Φ au milieu du déroulage de l'algorithme d'inférence. L'éclatement d'une relation dans une telle situation pourrait faire perdre en polymorphisme.

Par ailleurs, pour définir la relation d'ordre entre schémas de type que nous allons utiliser par la suite, nous allons classifier, dans un ensemble de contraintes, les variables de type dans deux catégories exclusives que nous nommerons :

- Les variables de type « intermédiaires ».
- Les variables de type « arguments ».

Définition 12 (Variable de type intermédiaire)**.**

Dans un ensemble de contraintes Φ*, une variable de type* α *est dite « intermédiaire » s'il n'existe aucune contrainte* $C \in \Phi$ *dans laquelle* α *apparaît en argument d'un type* `t`*.*

Dans un ensemble de contraintes Φ, les variables de type « argument » sont les variables qui ne sont pas des variables de type « intermédiaires ».

La définition suivante de la relation d'ordre entre schémas de type spécifie le minimum d'éclatement autorisé permettant d'énoncer et de démontrer des lemmes corrects de réduction du sujet.

Définition 9 (Relation d'ordre sur les schémas de type)**.**

Deux schémas de type $[\ \forall\alpha_1 \dots \alpha_n\ .\ \alpha_0 \mid \Phi\ ]$ *et* $[\ \forall\alpha'_1 \dots \alpha'_{n'}\ .\ \alpha'_0 \mid \Phi'\ ]$ *vérifient la relation* $[\ \forall\alpha_1 \dots \alpha_n\ .\ \alpha_0 \mid \Phi\ ] \leqslant [\ \forall\alpha'_1 \dots \alpha'_{n'}\ .\ \alpha'_0 \mid \Phi'\ ]$ *s'il existe une fonction totale de renommage* R *des variables de type de telle sorte que :*

- $R(\alpha_0) = \alpha'_0$
- $R(\{\ \alpha_1, \dots, \alpha_n\ \}) \subset \{\ \alpha'_1, \dots, \alpha'_{n'}\ \}$
- $\forall\alpha\ .\ \alpha \notin \{\ \alpha_1, \dots, \alpha_n\ \} \Rightarrow R(\alpha) = \alpha$
- *Pour tout* $C \in \Phi$*, au moins l'une des propriétés suivantes est vérifiée :*
  - $R(C) \in \Phi'$
  - C *est de la forme* $\sigma \leqslant \alpha$ *et il existe* $C' \in \Phi'$ *de la forme* $\sigma' \leqslant R(\alpha)$ *tel que* $R(\sigma) \leqslant \sigma'$
  - C *est de la forme* $\tau^l \leqslant \alpha$*, la variable* α *est généralisée (i.e.* $\alpha \in \{\ \alpha_1, \dots, \alpha_n\ \}$*), la variable* $R(\alpha)$ *(qui est obligatoirement généralisée) est une variable de type intermédiaire dans* Φ' *et* $R([\ \forall\alpha'\ .\ \alpha' \mid \tau^l \leqslant \alpha'\ ]) \leqslant [\ \forall\alpha'_1 \dots \alpha'_{n'}\ .\ \alpha'_0 \mid \Phi'\ ]$ *avec* α' *une variable de type non-libre dans* τ^l*.*

Autrement dit, pour qu'un schéma de type $[\ \forall\alpha_1 \dots \alpha_n\ .\ \alpha_0 \mid \Phi\ ]$ soit plus petit qu'un autre schéma de type $[\ \forall\alpha'_1 \dots \alpha'_{n'}\ .\ \alpha'_0 \mid \Phi'\ ]$, comme précédemment, nous autorisons Φ' à contenir des contraintes absentes de Φ ainsi que des contraintes répliquées dans Φ (grâce à la non-injectivité de R). De plus, nous autorisons les contraintes de Φ' de la forme $\sigma \leqslant \alpha'$ à apparaître dans Φ avec un σ plus petit. Enfin, les contraintes de Φ' de la forme $\sigma \leqslant \alpha'$, avec α' généralisé et intermédiaire, peuvent être éclatées dans Φ.

Nous remarquerons qu'il est important d'imposer que $R(\alpha)$ soit intermédiaire dans Φ' et pas que α soit intermédiaire dans Φ. Ces deux propriétés ne sont pas équivalentes, et en particulier, la relation que nous aurions définie en imposant la deuxième propriété ne serait pas une relation d'ordre car la transitivité ne serait pas vérifiée.

Grâce à cette dernière définition, nous allons maintenant définir une relation d'ordre sur les ensembles de contraintes :

Définition 5 (Relation d'ordre sur les ensembles de contraintes)**.**

Les deux ensembles de contraintes Φ et Φ' vérifient $\Phi \leqslant \Phi'$ lorsque, pour tout $C \in \Phi$, au moins l'une des propriétés suivantes est vérifiée :

- $R(C) \in \Phi'$
- *C est de la forme $\sigma \leqslant \alpha$ et il existe $C' \in \Phi'$ de la forme $\sigma' \leqslant R(\alpha)$ tel que $R(\sigma) \leqslant \sigma'$*
- *C est de la forme $\tau^l \leqslant \alpha$, la variable $R(\alpha)$ est une variable de type intermédiaire dans Φ' et $R([\ \forall\alpha' \,.\, \alpha' \mid \tau^l \leqslant \alpha'\ ]) \leqslant \sigma'$ avec α' une variable de type non-libre dans τ^l.*

Le lemme de réduction du sujet se formule alors de la manière suivante :

Lemme 5 (Réduction du sujet par $(\longrightarrow)$)**.**

Soient :

- *e_1 et e_2 deux expressions vérifiant $e_1 \longrightarrow e_2$*
- *α une variable de type*
- *Φ un ensemble de contraintes valide et saturé contenant au maximum une contrainte de la forme $(\alpha \leqslant \tau^r)$ avec τ^r un type construit.*
- *Φ'_1 l'ensemble de contraintes obtenu par typage de $e_1 : \alpha$ dans $(\Phi, \varnothing)$ en dérivant l'arbre d'inférence au dessus de $\Phi, \varnothing \vdash e_1 : \alpha \rhd \Phi'_1$.*

Alors le typage de $e_2 : \alpha$ dans $(\Phi, \varnothing)$ réussit et l'ensemble de contraintes Φ'_2 obtenu en dérivant l'arbre d'inférence au dessus de $\Phi, \varnothing \vdash e_2 : \alpha \rhd \Phi'_2$ vérifie $\Phi'_2 \leqslant \Phi'_1$.

Démonstration (Lemme 5) :

Comme précédemment, nous raisonnons par disjonction des cas sur la définition de la fonction $(\longrightarrow)$. Les deux cas de réduction du `if` sont triviaux et très similaires à ceux des chapitres précédents. Les cas du `let` et du filtrage de motifs n'ont plus lieu ici. Les seuls cas pathologiques sont l'application d'une fonction et l'application d'une primitive (unaire ou binaire). Ces trois cas étant très similaires, nous n'analyserons ici que le cas de la réduction du λ :

- Cas « `(λ x . e) v` $\longrightarrow$ `e[x ↦ v]` » :
  Comme habituellement, nous cherchons à montrer qu'il est possible de construire l'arbre d'inférence de $\Phi, \varnothing \vdash$ `e[x ↦ v]` $: \alpha \rhd \Phi'_2$ à partir de l'arbre d'inférence de $\Phi, \varnothing \vdash$

$(\lambda\ \mathtt{x}\ .\ \mathtt{e})\ \mathtt{v} : \alpha \rhd \Phi'_1$ qui est obligatoirement de la forme :

$$\dfrac{\dfrac{\dfrac{}{\Phi_0 \vdash \alpha_1 \leqslant \alpha_3 \to \alpha_4 \rhd \Phi_1}\,T_1 \qquad \dfrac{}{\Phi_1 \vdash \alpha_4 \leqslant \alpha \rhd \Phi_2}\,T_2}{\Phi_2, \{\, \mathtt{x} : \alpha_5 \,\} \vdash \mathtt{e} : \alpha_6 \rhd \Phi_3}\,T_3 \qquad \dfrac{}{\Phi_3 \vdash \alpha_5 \to \alpha_6 \leqslant \alpha_1 \rhd \Phi_4}\,T_4}{\Phi_2, \varnothing \vdash \lambda\ \mathtt{x}\ .\ \mathtt{e} : \alpha_1 \rhd \Phi_4}\,\textsc{TLambda}$$

$$\dfrac{\dfrac{\dfrac{}{\Phi_4, \varnothing \vdash \mathtt{v} : \alpha_2 \rhd \Phi_5}\,T_5 \qquad \dfrac{}{\Phi_5 \vdash \textsc{gen}(\alpha_2, \Phi_5, \varnothing) \leqslant \alpha_3 \rhd \Phi'_1}\,T_6}{\Phi_0, \varnothing \vdash (\lambda\ \mathtt{x}\ .\ \mathtt{e})\ \mathtt{v} : \alpha \rhd \Phi'_1}\,\textsc{TApp}$$

Nous commençons alors par montrer qu'il est possible de construire un arbre d'inférence au dessus de $\Phi_0, \varnothing \vdash \mathtt{e}[\mathtt{x} \mapsto \mathtt{v}] : \alpha_6 \rhd \Phi_6$ en remplaçant les sous-arbres de T_3 correspondant au typage des occurrences libres de $\mathtt{x}$ dans $\mathtt{e}$ par une adaptation de T_5 :

$$\dfrac{\dfrac{}{\Phi_0, \varnothing \vdash \mathtt{e}[\mathtt{x} \mapsto \mathtt{v}] : \alpha_6 \vdash \Phi_6}\,T'_3}{\mathtt{v} : \alpha'}\,T'_5 \qquad \mathtt{v} : \alpha''\,T''_5$$

Il faut néanmoins montrer que l'ensemble de contraintes Φ_6 vérifie bien $\Phi_6 \leqslant \Phi'_1$. Pour ce faire, nous analysons les modifications apportées dans l'ensemble de contraintes généré par le remplacement des sous-arbres de T_3 correspondant aux différentes instances libres de la variable $\mathtt{x}$ dans $\mathtt{e}$ par l'arbre d'inférence de $\mathtt{v}$. Pour chaque occurrence de $\mathtt{x}$, il faut distinguer trois cas exclusifs en fonction de sa position dans l'expression :

- Soit $\mathtt{x}$ est en position d'être « déconstruit », directement ou indirectement. Dans la version du langage étudiée dans ce chapitre, cela revient à dire que $\mathtt{x}$ est soit en position de fonction dans un appel, soit en position de test dans un `if-then-else`. Par « indirectement », nous entendons qu'il peut également être dans le `then` ou le `else` d'un `if-then-else` lui même placé en position d'être déconstruit. Par exemple, dans l'expression `((if true then` λ `y . y else x) 42)`, la variable $\mathtt{x}$ est

en position d'être déconstruite.

Dans cette situation, nous sommes sûr que le typage de cette occurrence de `x` dans l'expression `e` avec la variable de type α' avait engendré une et une seule comparaison de la forme $\alpha' \leqslant \tau^r$ avec τ^r un type construit (une flèche dans le cas d'une application et `bool` dans le cas `if-then-else`).

Grâce aux contraintes $(\textsc{gen}(\alpha_2, \Phi_4, \varnothing) \leqslant \alpha_3)$ et $(\alpha_3 \leqslant \alpha_5)$ (dû à $(\alpha_5 \to \alpha_6 \leqslant \alpha_1)$ et $(\alpha_1 \leqslant \alpha_3 \to \alpha_4)$), l'ajout de la contrainte $(\alpha' \leqslant \tau^r)$ avait engendré une et une seule instanciation du schéma $\textsc{gen}(\alpha_2, \Phi_5, \varnothing)$, et le typage de `v` à la place de cette occurrence de `x` dans `e` génère après ce pas d'évaluation au plus les mêmes contraintes (à α-renommage près) que celles introduites lors de cette instanciation.

- Soit `x` est argument (direct ou indirect) dans un appel de fonction ou de primitive. De la même manière que dans le cas précédent, nous entendons par « indirect » le fait que `x` puisse être dans le `then` ou le `else` d'un `if-then-else` lui-même argument (direct ou indirect) dans un appel.

  Dans cette situation, une instance de la règle TApp avait engendré une généralisation de α' lors du typage de `x` : α' dans l'expression `e` originale. Cette généralisation avait provoqué la création d'un schéma de type de la forme $[\ \forall\alpha'_1 \dots \alpha'_n\ .\ \alpha' \mid \Phi'\ ]$ dans lequel l'ensemble de contraintes Φ' contenait la contrainte $(\textsc{gen}(\alpha_2, \Phi_5, \varnothing) \leqslant \alpha')$.

  Maintenant que `x` est remplacé par `v` dans `e`, les contraintes qui étaient avant regroupées dans $\textsc{gen}(\alpha_2, \Phi_5, \varnothing)$ sont maintenant éclatées dans le schéma de type généré par cette utilisation de TApp. D'après la définition des règles de typage, la variable α' ne peut pas apparaître dans Γ. En effet, seule la règle TVar extrait une variable de type de Γ et a en prémisse une conclusion de règle de saturation et non une conclusion de règle de typage. La variable α' est alors bien généralisée (ce que l'on peut résumer par $\alpha' \in \{\ \alpha'_1, \dots, \alpha'_n\ \}$).

  Nous somme alors dans le tout dernier cas de la définition de la relation d'ordre entre deux ensembles de contraintes. L'ensemble de contraintes généré par l'application de TApp est donc effectivement plus petit après évaluation d'un pas que l'ensemble de contraintes original.

- Soit `x` est « résultat » (direct ou indirect) de l'expression. Dans cette situation, la variable de type α' n'est pas comparée à un type construit, mais uniquement à d'éventuelles variables de type intermédiaires et en particulier à α via la contrainte $\alpha' \leqslant \alpha$. Grâce aux contraintes $(\textsc{gen}(\alpha_2, \Phi_5, \varnothing) \leqslant \alpha_3)$, $(\alpha_3 \leqslant \alpha_5)$ et $(\alpha_5 \leqslant \alpha')$, le typage de l'expression originale avait alors engendré la contrainte $(\textsc{gen}(\alpha_2, \Phi_5, \varnothing) \leqslant \alpha)$.

  Maintenant que la variable `x` est remplacée par `v` dans `e`, les contraintes engendrées par le typage de `v` sont éclatées dans l'ensemble de contraintes Φ_6 et accumulées sur α plutôt que d'être regroupées dans la contrainte $(\textsc{gen}(\alpha_2, \Phi_5, \varnothing) \leqslant \alpha)$.

  Nous sommes une nouvelle fois dans le dernier cas de la définition de la relation d'ordre entre deux ensembles de contraintes.

Dans tous les cas, l'ensemble de contraintes Φ_6 généré par typage de l'expression après

un pas d'évaluation vérifie bien $\Phi_6 \leqslant \Phi'_1$.
Grâce aux contraintes $(\alpha_5 \to \alpha_6 \leqslant \alpha_1)$ (ajouté dans T_2) et $(\alpha_1 \leqslant \alpha_3 \to \alpha_4)$ (ajouté dans T_4), le sous-arbre T_4 montre obligatoirement la compatibilité de la contrainte $(\alpha_6 \leqslant \alpha_4)$ avec les autres. Puisque T_2 a ajouté la contrainte $(\alpha_4 \leqslant \alpha)$, le sous-arbre T_4 montre également la compatibilité de $(\alpha_6 \leqslant \alpha)$.
Toutes les contraintes accumulées sur α_6 dans les différents sous-arbres ont donc bien été prouvées compatibles avec α également. Par ailleurs, l'ensemble de contraintes Φ_2 ne diffère de Φ que par l'ajout des deux contraintes $(\alpha_1 \leqslant \alpha_3 \to \alpha_4)$ et $(\alpha_4 \leqslant \alpha)$. La construction de l'arbre d'inférence au dessus de $\Phi, \varnothing \vdash$ `[x ↦ v]` $: \alpha \rhd \Phi'_2$ est alors possible et ne génère pas plus de contraintes que dans Φ_6 (à α-renommage près). La relation $\Phi'_2 \leqslant \Phi'_1$ est en particulier bien vérifiée.

La seule autre difficulté de la preuve de validité de ce nouveau système de types concerne le passage au contexte du lemme précédent :

Lemme 6 (Réduction du sujet par $(\longmapsto)$)**.**

Soient :

- e_1 *et* e_2 *deux expressions vérifiant* $e_1 \longmapsto e_2$
- α *une variable de type*
- Φ *un ensemble de contraintes valide et saturé contenant au maximum une contrainte de la forme* $(\alpha \leqslant \tau^r)$ *avec* τ^r *un type construit.*
- Φ'_1 *l'ensemble de contraintes obtenu par typage de* $e_1 : \alpha$ *dans* $(\Phi, \varnothing)$ *en dérivant l'arbre d'inférence au dessus de* $\Phi, \varnothing \vdash e_1 : \alpha \rhd \Phi'_1$.

Alors le typage de $e_2 : \alpha$ *dans* $(\Phi, \varnothing)$ *réussit et l'ensemble de contraintes* Φ'_2 *obtenu en dérivant l'arbre d'inférence au dessus de* $\Phi, \varnothing \vdash e_2 : \alpha \rhd \Phi'_2$ *vérifie* $\Phi'_2 \leqslant \Phi'_1$.

Démonstration (Lemme 6) :

Ce lemme est plus difficile à démontrer que son équivalent dans les systèmes précédents. Il n'est en effet plus possible d'exprimer les lemmes de monotonie qui nous simplifiaient grandement ce genre de démonstration jusqu'ici.
Nous démontrons ce lemme par induction sur la structure des expressions et analyse de cas sur la définition du contexte d'évaluation `E` :

- Cas « `[]` » :
  Conséquence directe du lemme 5.

- Cas « `E e` » :
  Nous souhaitons montrer qu'il est possible de construire l'arbre d'inférence de $\Phi, \varnothing \vdash$

$\mathtt{e}_2\ \mathtt{e} : \alpha \rhd \Phi'_2$ grâce à l'arbre d'inférence de $\Phi, \varnothing \vdash \mathtt{e}_1\ \mathtt{e} : \alpha \rhd \Phi'_1$ sachant que $\mathtt{e}_1 \longmapsto \mathtt{e}_2$. D'après la structure de l'expression (une application), la règle de typage appliquée à la racine de ces arbres est obligatoirement TApp :

$$\textsc{TApp}\ \dfrac{\dfrac{}{\Phi \vdash \alpha_1 \leqslant \alpha_3 \to \alpha_4 \rhd \Phi_1}\,T_1 \quad \dfrac{}{\Phi_1 \vdash \alpha_4 \leqslant \alpha \rhd \Phi_2}\,T_2 \quad \dfrac{}{\Phi_2, \varnothing \vdash \mathtt{e}_1 : \alpha_1 \rhd \Phi_3}\,T_3 \quad \dfrac{}{\Phi_3, \varnothing \vdash \mathtt{e} : \alpha_2 \rhd \Phi_4}\,T_4 \quad \dfrac{}{\Phi_4 \vdash \textsc{gen}(\alpha_2, \Phi_4, \varnothing) \leqslant \alpha_3 \rhd \Phi'_1}\,T_5}{\Phi, \varnothing \vdash \mathtt{e}_1\ \mathtt{e} : \alpha \rhd \Phi'_1}$$

L'arbre d'inférence obtenu après remplacement de e_1 par e_2 est similaire :

$$\textsc{TApp}\ \dfrac{\dfrac{}{\Phi \vdash \alpha_1 \leqslant \alpha_3 \to \alpha_4 \rhd \Phi_1}\,T_1 \quad \dfrac{}{\Phi_1 \vdash \alpha_4 \leqslant \alpha \rhd \Phi_2}\,T_2 \quad \dfrac{}{\Phi_2, \varnothing \vdash \mathtt{e}_2 : \alpha_1 \rhd \Phi'_3}\,T'_3 \quad \dfrac{}{\Phi'_3, \varnothing \vdash \mathtt{e} : \alpha_2 \rhd \Phi'_4}\,T'_4 \quad \dfrac{}{\Phi'_4 \vdash \textsc{gen}(\alpha_2, \Phi'_4, \varnothing) \leqslant \alpha_3 \rhd \Phi'_2}\,T'_5}{\Phi, \varnothing \vdash \mathtt{e}_1\ \mathtt{e} : \alpha \rhd \Phi'_2}$$

La variable α_1 étant fraîche, l'unique contrainte de la forme $\alpha_1 \leqslant \tau^r$ est celle ajoutée dans T_1. Nous en déduisons que toutes les contraintes de la forme $\sigma \leqslant \alpha_1$ engendrées par le typage de $e_1 : \alpha_1$ avaient été instanciées une et une seule fois dans T_3 lors de leur comparaison avec $\alpha_3 \to \alpha_4$. Le fait que ces contraintes de la forme $\sigma \leqslant \alpha_2 \to \alpha_4$ soient éclatées après remplacement de e_1 par e_2 n'engendre pas plus de contraintes que précédemment puisque les instanciations avaient généré ces mêmes contraintes (à α-renommage près).

De plus, la variable α_1 est une variable de type intermédiaire (i.e. elle n'apparaît pas comme argument d'un type `t`). Le fait que les contraintes de la forme $\sigma \leqslant \alpha_1$ soient maintenant éclatées est compatible avec la décroissance de l'ensemble de contraintes.

Par ailleurs, α_2 étant fraîche, le typage de $e : \alpha_2$ ayant réussi initialement (en produisant T_4) réussit à nouveau en produisant un arbre d'inférence T'_4 de même structure que T_4. Seuls les ensembles de contraintes propagées dans ce sous-arbre changent : ils sont plus petits.

Enfin, l'ensemble de contraintes Φ'_4 étant plus petit que Φ_4, nous avons $\textsc{gen}(\alpha_2, \Phi'_4, \varnothing) \leqslant \textsc{gen}(\alpha_2, \Phi_4, \varnothing)$. Le remplacement de Φ_4 par Φ'_4 dans T'_5 ne remet donc pas en cause la

décroissance de l'ensemble de contraintes (nous sommes ici dans l'avant-dernier cas de la définition de la relation d'ordre sur les ensembles de contraintes).

- Cas « `v E` » :
  Nous raisonnons de manière similaire. Nous souhaitons montrer qu'il est possible de construire l'arbre d'inférence de $\Phi, \varnothing \vdash \mathtt{v}\ \mathtt{e}_2 : \alpha \rhd \Phi'_2$ grâce à l'arbre d'inférence de $\Phi, \varnothing \vdash \mathtt{v}\ \mathtt{e}_1 : \alpha \rhd \Phi'_1$ sachant que $\mathtt{e}_1 \longmapsto \mathtt{e}_2$. D'après la structure de l'expression (une application), la règle de typage appliquée à la racine de ces arbres est obligatoirement TApp :

$$\textsc{TApp}\ \dfrac{\dfrac{\overset{T_1}{\triangledown}}{\Phi \vdash \alpha_1 \leqslant \alpha_3 \to \alpha_4 \rhd \Phi_1} \quad \dfrac{\overset{T_2}{\triangledown}}{\Phi_1 \vdash \alpha_4 \leqslant \alpha \rhd \Phi_2} \quad \dfrac{\overset{T_3}{\triangledown}}{\Phi_2, \varnothing \vdash \mathtt{v} : \alpha_1 \rhd \Phi_3} \qquad \dfrac{\overset{T_4}{\triangledown}}{\Phi_3, \varnothing \vdash \mathtt{e}_1 : \alpha_2 \rhd \Phi_4} \quad \dfrac{\overset{T_5}{\triangledown}}{\Phi_4 \vdash \textsc{gen}(\alpha_2, \Phi_4, \varnothing) \leqslant \alpha_3 \rhd \Phi'_1}}{\Phi, \varnothing \vdash \mathtt{e}_1\ \mathtt{e} : \alpha \rhd \Phi'_1}$$

  L'arbre d'inférence obtenu après remplacement de e_1 par e_2 est similaire :

$$\textsc{TApp}\ \dfrac{\dfrac{\overset{T_1}{\triangledown}}{\Phi \vdash \alpha_1 \leqslant \alpha_3 \to \alpha_4 \rhd \Phi_1} \quad \dfrac{\overset{T_2}{\triangledown}}{\Phi_1 \vdash \alpha_4 \leqslant \alpha \rhd \Phi_2} \quad \dfrac{\overset{T_3}{\triangledown}}{\Phi_2, \varnothing \vdash \mathtt{v} : \alpha_1 \rhd \Phi_3} \qquad \dfrac{\overset{T'_4}{\triangledown}}{\Phi_3, \varnothing \vdash \mathtt{e}_2 : \alpha_2 \rhd \Phi'_4} \quad \dfrac{\overset{T'_5}{\triangledown}}{\Phi'_4 \vdash \textsc{gen}(\alpha_2, \Phi'_4, \varnothing) \leqslant \alpha_3 \rhd \Phi'_2}}{\Phi, \varnothing \vdash \mathtt{e}_1\ \mathtt{e} : \alpha \rhd \Phi'_2}$$

  La variable α_2 est fraîche. Nous savons donc, par hypothèse d'induction, que T'_4 est constructible et que $\Phi'_4 \leqslant \Phi_4$. Nous en déduisons $\textsc{gen}(\alpha_2, \Phi'_4, \varnothing) \leqslant \textsc{gen}(\alpha_2, \Phi_4, \varnothing)$ et concluons que T'_5 est constructible et que $\Phi'_2 \leqslant \Phi'_1$.

- Cas « $\mathtt{p}^1$ `E` », « $\mathtt{p}^2$ `E e` » et « $\mathtt{p}^2$ `v E` » :
  Très similaires au cas précédent.

- Cas « `if E then` $\mathtt{e}_3$ `else` $\mathtt{e}_4$ » :
  La structure de l'arbre de typage initial est :

$$\textsc{TIF}\ \dfrac{\dfrac{\overset{T_1}{\triangledown}}{\Phi \vdash \alpha' \leqslant \mathtt{bool} \rhd \Phi_1} \quad \dfrac{\overset{T_2}{\triangledown}}{\Phi_1, \varnothing \vdash \mathtt{e}_1 : \alpha' \rhd \Phi_2} \quad \dfrac{\overset{T_3}{\triangledown}}{\Phi_2, \varnothing \vdash \mathtt{e}_3 : \alpha \rhd \Phi_3} \quad \dfrac{\overset{T_4}{\triangledown}}{\Phi_3, \varnothing \vdash \mathtt{e}_4 : \alpha \rhd \Phi'_1}}{\Phi, \varnothing \vdash \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_3\ \mathtt{else}\ \mathtt{e}_4 : \alpha \rhd \Phi'_1}$$

  et après remplacement de e_1 par e_2 :

$$\textsc{TIF}\ \dfrac{\dfrac{\overset{T_1}{\triangledown}}{\Phi \vdash \alpha' \leqslant \mathtt{bool} \rhd \Phi_1} \quad \dfrac{\overset{T'_2}{\triangledown}}{\Phi_1, \varnothing \vdash \mathtt{e}_2 : \alpha' \rhd \Phi''_2} \quad \dfrac{\overset{T'_3}{\triangledown}}{\Phi''_2, \varnothing \vdash \mathtt{e}_3 : \alpha \rhd \Phi''_3} \quad \dfrac{\overset{T'_4}{\triangledown}}{\Phi''_3, \varnothing \vdash \mathtt{e}_4 : \alpha \rhd \Phi'_2}}{\Phi, \varnothing \vdash \mathtt{if}\ \mathtt{e}_2\ \mathtt{then}\ \mathtt{e}_3\ \mathtt{else}\ \mathtt{e}_4 : \alpha \rhd \Phi'_2}$$

  La variable de type α' est fraîche. L'unique contrainte générée de la forme $\alpha' \leqslant \tau^r$ est $\alpha' \leqslant \mathtt{bool}$. Chaque contrainte de la forme $\sigma \leqslant \alpha'$ générée lors du typage de e_1 a donc engendré une et une seule instanciation dans l'arbre initial lors de la rencontre de σ avec `bool`. L'éclatement potentiel de ces σ lors du passage de e_1 à e_2 n'engendre donc aucune perte en polymorphisme.
  Le typage de e_3 et de e_4 est indépendant. Il n'est pas impacté par la modification des contraintes sur α'.

4.7 Signification de la taille limite des clés

La borne sur la taille des clés est un paramètre global du système de types. Ce que nous présentons dans ce chapitre peut donc être vu comme une « suite » de systèmes de types de plus en plus puissants de par leur capacité à gérer le polymorphisme. Le programmeur peut donc paramétrer son système de types en fonction de son besoin en polymorphisme.

Nous allons voir que l'interprétation de ce paramètre est assez intuitive :

- En définissant ce paramètre à 0, nous obtenons un système de types valide mais sans polymorphisme. En effet, tous les schémas de type sont alors intanciés avec la même clé vide, et toutes les instances d'un même schéma mentionnent donc les mêmes variables de type. Dans ce contexte, les généralisations sont inutiles.
- En définissant ce paramètre à 1, nous obtenons un système de types dans lequel une valeur polymorphe est capable de subir une réplication à travers le paramètre d'un λ avant d'être utilisée dans des contexte de typage différents dans le corps du λ en question.

  Ainsi, l'expression :

  ```
  (λ id . (id 42, id "hello")) (λ x . x)
  ```

  sera acceptée par un typeur muni de polymorphisme de niveau 1, et son schéma de type associé sera bien celui attendu :

  $$[\ \forall\alpha_1\alpha_2\alpha_3\ .\ \alpha_1 \mid \alpha_2 \times \alpha_3 \leqslant_{[]} \alpha_1\ \wedge\ \texttt{int} \leqslant_{[0]} \alpha_2\ \wedge\ \texttt{string} \leqslant_{[1]} \alpha_3\ ]$$

  Les clés présentes dans ce schéma ne sont, dans ce cas particulier, que des résidus de l'algorithme de typage.

  La différence importante avec un mécanisme d'inférence fournissant du polymorphisme classique (via un typage spécifique du `let`) est que nous n'imposons pas qu'une fonction, dont le paramètre est utilisé dans différents « contextes », soit appliquée immédiatement à un argument polymorphe, et donc que l'expression soit de la forme $(\texttt{(}\lambda\ \texttt{x . } e_2\texttt{)}\ e_1\texttt{)}$. La fonction en question peut maintenant être définie indépendamment de son usage, et même dans un autre « module » (pour peu que notre langage en soit muni). L'« interface » du module posséderait alors, via les clés indiçant les relations de sous-typage, toutes les informations nécessaires à l'usage des fonctions exportées avec des arguments polymorphes.

  C'est cette propriété du nouveau système qui permet à un tel typeur muni de polymorphisme de niveau 1 d'accepter les exemples relatifs aux objets que nous avons mentionnés précédemment, pour peu que nous mixions l'extension de ce chapitre avec celle du chapitre précédent.

  Nous verrons néanmoins, grâce au prochain exemple, que le polymorphisme de niveau 1 n'est pas toujours aussi puissant que le polymorphisme classique.
- Certaines expressions nécessitent un polymorphisme de niveau supérieur pour être acceptées, comme par exemple :

  ```
  (fst ((λf.(f 3, f "hello")) (fst ((λg.(g, g)) (λx.x))))) + 1
  ```

  qui n'est typable qu'avec un polymorphisme de niveau 2 ou plus. En effet, la fonction polymorphe `(λ x . x)` doit subir deux réplications (une à travers `g` puis une à travers `f`) avant d'être utilisée dans les deux contextes `(f 3)` et `(f "hello")`.

Cette expression est néanmoins équivalente à :

```
fst (
  let f =
    fst (
      let g = λ x . x in
      (g, g)
    ) in
    (f 3, f "hello")
) + 1
```

qu'un système de types muni d'un mécanisme de généralisation classique (mais sans « value-restriction ») accepterait.

Certaines expressions plus complexes comme :

```
(λ f . f f) ((λ g . g g) (λ x . x)) 42
```

nécessitent un polymorphisme de niveau au moins 4. La valeur polymorphe `(λ x . x)` doit en effet traverser les quatre usages des variables `f` et `g` avant d'être appliquée à 42.

L'implémentation de ce système, réalisée dans le cadre de cette thèse, calcule par dichotomie, lorsqu'il est compris dans des bornes raisonnables (entre 1 et 2^{16}), le niveau de polymorphe minimal nécessaire à l'acceptation d'une expression donnée. Elle peut donc être pratique pour tester le niveau de polymorphisme minimal nécessaire à différentes expressions. Sur des exemples de code « standard », le niveau de polymorphisme minimal est souvent de 0 ou 1, parfois de 2.

Il existe malheureusement des situations ou l'extension présentée dans ce chapitre n'est pas suffisante, typiquement pour gérer la récursion polymorphe. C'est le cas de la fonction `eval` sur des AST, exemple très classique d'application des **GADT**. Pour gérer ce genre de situation, nous devrons faire appel à des techniques plus poussées à base de types existentiels. C'est l'objet du prochain chapitre.

Chapitre 5

Types existentiels et GADT

Nous nous intéressons, dans ce chapitre, à l'extension de notre système de types de base avec une variante des **GADT**. Pour prendre en charge cette nouvelle construction, nous sommes amenés à définir une variante de la notion classique de « type existentiel » permettant de dénoter un ensemble inconnu de valeurs (tout comme une variable de type universelle classique) mais qui n'est pas « quelconque », ce qui se traduit par le fait qu'il est interdit de lui imposer certaines contraintes de sous-typage.

Dans un système à base d'unification, les types existentiels sont habituellement manipulés de manière similaire aux types abstraits. Dans nos systèmes à base de sous-typage, il serait bien entendu possible de les définir de la même manière en interdisant toute relation de sous-typage entre un type existentiel et un type construit. Néanmoins, nous préférons changer radicalement de vision sur ce point pour bénéficier du côté asymétrique de la relation de sous-typage et de la présence d'un langage riche de contraintes de types.

Par ailleurs, nous souhaitons maintenir au maximum la capacité d'inférence de notre système de types et ainsi nécessiter le minimum d'annotations de type. Pour ce faire, nous étendons les mécanismes introduits dans le chapitre 3 de cette thèse. À cette occasion, nous avions en particulier défini un opérateur, noté (#), signifiant qu'« un type n'est pas compatible avec un constructeur ». Dans le contexte plus général de ce chapitre, les contraintes dont nous souhaitons manipuler le « contraire » ne porteront plus uniquement sur des noms de constructeurs de données, mais sont des contraintes de sous-typage quelconques. Nous sommes alors amenés à ajouter un opérateur générique de « négation » dans notre langage de contraintes en plus des opérateurs de conjonction et de disjonction déjà introduits dans les chapitres précédents. Les mécanismes de saturation de contraintes doivent alors gérer des formules logiques du premier ordre reliant des contraintes de sous-typage.

Nous commençons par donner l'intuition de notre approche des **GADT**et des techniques que nous avons développées pour les gérer (section 5.2). Nous les formalisons ensuite par la définition d'un système de types (section 5.3) dont nous prouverons la terminaison (section 5.4) et la correction (section 5.5).

La seule situation dans laquelle l'inférence reste impossible en général dans notre système concerne la récursion polymorphe. Nous avons alors recours à une annotation de type. Nous montrons à la fin de ce chapitre (section 5.6) que la récursion polymorphe peut être gérée par un mécanisme de typage très similaire à celui que nous définissons pour les **GADT**.

5.1 Contexte

Les « types existentiels » comme l'ont montré théoriquement Läufer et Odersky en 1992 (cf. [LO92]) puis Mauny et Pottier dans la pratique en 1994 (cf. [MP94]), sont une construction intéressante et utile pour la programmation en général et l'écriture de bibliothèques génériques en particulier.

Dans leur version de base, les types existentiels ne nécessitent pas d'extension du langage des types. Ils sont simplement vus comme une interprétation particulière des variables de type libres, habituellement interdites en ML classique. Leur manipulation lors du typage nécessite toutefois un traitement spécifique non-trivial, mais qui reste décidable grâce à des annotations de type (cf. [PR06, SP09]).

Elles ont été popularisées grâce à une nouvelle construction : les « types algébriques gardés » ou `GADT`, que l'on retrouve maintenant dans différents langages comme Haskell (cf. [SP09]) et OCaml (cf. [GL11]). Il s'agit d'une simple extension des types algébriques standard dans laquelle les constructeurs de données possèdent une annotation permettant de lier librement les paramètres du constructeur de types avec les types des arguments du constructeur. Ces liaisons sont en particulier autorisées à faire intervenir des variables de type libres.

Dans un système de type à base d'unification, comme le sont la plupart des systèmes implémentant des `GADT` à l'heure actuelle, la notion de variable rigide est claire : il s'agit de variables sur lesquelles on ne sait rien, et qui sont donc en général manipulées comme des types abstraits, tous différents les uns des autres et différents de tout autre type.

En présence de sous-typage, la notion de variable rigide se complexifie et de nouvelles questions se posent, en particulier au niveau de leur variance comme le font remarquer Scherer et Remy dans leur article de 2013 (cf. [SR13]).

En réalité, la définition même de la notion de « variable rigide » doit être remise en cause pour s'adapter au contexte du sous-typage. Pour en bénéficier au maximum, il ne faut en particulier plus considérer les variables rigides comme simplement différentes des autres types, mais autoriser l'existence de liens à la fois entre elles, avec des variables universelles et des types paramétrés. C'est à ce problème que nous nous attaquons dans ce chapitre.

5.2 Concepts de base

Nous décrivons ici les problèmes que nous abordons dans ce chapitre et donnons les « idées fondamentales » du système de types que nous allons définir formellement par la suite. Le discours se veut assez informel car le but de cette section est uniquement de donner l'intuition des mécanismes de typage que nous allons mettre en place dans la section suivante.

5.2.1 Objectif

Il est parfois intéressant, en programmation, d'encapsuler des données en « masquant » certaines informations de typage les concernant. Un exemple typique, peu utile en pratique mais représen-

tatif de ce que l'on souhaite être capable de faire, consiste à retarder un appel. Pour ce faire, nous souhaitons stocker, dans une même « boîte », une fonction et un argument pour cette fonction.

Dans notre contexte, ces boîtes sont simplement des constructeurs de données. Concrètement, nous souhaitons accepter un code comme :

```
(* Définition du type des boîtes contenant une fonction      *)
(* et un argument pour cette fonction.                        *)
type box = Box (β → unit, β)

(* Définition de deux boîtes contenant des                    *)
(* fonctions/arguments de types différents.                   *)
let box1 = Box (print_string, "hello")
let box2 = Box (print_int, 42)

(* Accès aléatoire à l'une des deux boîtes.                   *)
let box = if rand () then box1 else box2

(* Ouverture de la boîte et appel de la fonction              *)
(* qu'elle contient.                                          *)
match box with Box (f, x) → f x
```

Toute la difficulté de la vérification du bon typage d'un tel code est concentrée sur l'ouverture de la boîte et la manipulation de son contenu. Nous devons en particulier être capable d'associer un « type » à la variable `x` de la dernière ligne et vérifier qu'il est « compatible » avec le « type » du paramètre de la fonction `f`.

Il n'est malheureusement pas suffisant de représenter ce « type » intermédiaire par une simple variable de type. En effet, rien n'empêcherait, a priori, de comparer cette variable de type avec un type construit (comme `float`, par exemple), alors qu'une telle comparaison devrait être interdite puisque `x` n'est pas obligatoirement un flottant. En d'autre termes, il faut être capable de refuser le code :

```
match box with Box (f, x) → print_float x
```

alors qu'a priori, aucune contrainte n'apparaît sur le type de `x`.

5.2.2 Déclaration des GADT

Comme nous l'avons suggéré précédemment, la présence de sous-typage dans nos systèmes nous incite à revoir la notion de « type existentiel ». En effet, l'unification nous incite habituellement a considérer qu'un type existentiel est complètement abstrait et qu'il est interdit de l'unifier avec un type construit quelconque. Dans notre contexte, lors de la définition d'un type « **GADT** »,

pour chaque constructeur de données, nous allons autoriser le programmeur à mentionner un ensemble de contraintes de sous-typage permettant de relier entre eux les éventuels paramètres du type **GADT**, les éventuelles variables de type locales au constructeur de données, et d'autres types classiques.

Nous étendons alors notre langage avec la possibilité de définir de nouveaux type **GADT**. En voici un exemple :

```
type α t =
  | P          [ unit ⩽ α ]
  | Q β        [ β ⩽ int ∧ int ⩽ α ]
  | R (β1, β2) [ β1 ⩽ β2 ∧ β2 ⩽ α ]
  | S β1       [ β1 ⩽ β2 → α ]
```

Nous classifions les variables de type apparaissant dans une telle définition en deux catégories exclusives :

- Les paramètres du type. Il n'y en a qu'un dans l'exemple précédent : le paramètre α. Lors du typage de la construction d'une valeur de type `t` et d'un filtrage sur un **GADT**, chacun de ces paramètres provoquera la génération d'une variable de type universelle classique.
- Les autres variables de type, à savoir les paramètres des constructeurs de données (comme par exemple β pour le constructeur de données `Q`) et les variables qui apparaissent dans les contraintes sans être ni un paramètre du type ni un paramètre d'un constructeur de données (comme par exemple le β_2 du constructeur de données `S`). Chacune de ces variables provoquera la génération d'une variable de type universelle lors du typage de la construction d'une valeur de type `t`, et d'un type existentiel lors de la déconstruction d'une valeur de type `t` via un filtrage.

Pour normaliser la définition formelle d'un **GADT** et simplifier les mécanismes de typage, nous interdisons un paramètre du type à apparaître en argument d'un constructeur de données. Ceci ne restreint en rien l'expressivité des **GADT** de notre langage. En effet, pour alléger l'écriture du code dans le contexte d'une implémentation d'un langage avec une syntaxe concrète, nous pourrons autoriser une définition comme :

```
type α t = K α
```

simplement en la macro-expansant en :

```
type α t = K β [ α ⩽ β ∧ β ⩽ α ]
```

Par ailleurs, nous remarquerons que les contraintes associées aux différents constructeurs de données sont « indépendantes » : il n'y a aucun lien entre les variables de type qu'ils mentionnent (en dehors des paramètres du type). Ainsi, la variable β_1 du constructeur `R` n'a aucun lien avec la variable β_1 du constructeur `S` et on pourrait les renommer indépendamment.

5.2.3 Typage du filtrage

Dans le chapitre 3 sur une extension du filtrage de motifs, une « disjonction de contraintes alternatives », notée Ψ, était propagée à travers les règles de typage et de saturation. Lors du typage d'un cas de filtrage sur un constructeur `K`, une astuce consistait à ajouter dans Ψ une relation de la forme $(\alpha_e \# \texttt{K})$ où α_e était la variable de type utilisée pour typer l'expression sur laquelle le filtrage était appliqué. Le mécanisme de saturation de contraintes ajoutait alors cette relation $(\alpha_e \# \texttt{K})$ dans toutes les disjonctions de contraintes induites par le typage du corps du cas en question. Cela permettait, en cas d'incompatibilité de ces contraintes, de ne pas provoquer de clash mais d'uniquement interdire ce cas de filtrage.

Dans ce chapitre, nous avons recours à un mécanisme similaire, mais au lieu d'utiliser les noms des constructeurs pour créer des relations de la forme $(\alpha_e \# \texttt{K})$, nous introduisons dans Ψ la négation de la contrainte associée au constructeur provenant de la définition du type **`GADT`** correspondant. Puisque cette contrainte peut contenir des disjonctions, sa négation peut contenir des conjonctions. Le Ψ représentant un ensemble de contraintes alternatives propagées à travers les règles d'inférence est maintenant une formule logique générale sous forme normale disjonctive, avec potentiellement des conjonctions.

Pour ce faire, nous devons étendre notre langage de contraintes avec un opérateur de disjonction (similaire à celui du chapitre 3) ainsi qu'un opérateur de négation. Pour des raisons pratiques, nous considérons toujours les contraintes de sous-typage sous leur forme normale conjonctive ou disjonctive en fonction des situations. Les négations sont en particulier toujours redescendues au niveau des relations de sous-typage :

$$
\begin{array}{lcl}
\mathsf{C} & ::= & \tau^l \leqslant \tau^r \mid \tau^r \not\leqslant \tau^l \\
\psi & ::= & \mathsf{C}_1 \vee \ldots \vee \mathsf{C}_n \\
\Phi & ::= & \psi_1 \wedge \ldots \wedge \psi_n \\
\phi & ::= & \mathsf{C}_1 \wedge \ldots \wedge \mathsf{C}_n \\
\Psi & ::= & \phi_1 \vee \ldots \vee \phi_n
\end{array}
$$

Ainsi, lors du typage d'un cas de filtrage de la forme « `K x` $\rightarrow$ `e` », nous insérons simplement dans l'ensemble de contraintes alternatives Ψ la négation de la contrainte associée au constructeur `K` lors de sa déclaration.

Pour illustrer ce mécanisme, considérons l'exemple simple suivant :

```
type α t =
  | I [ α ⩽ int ]
  | S [ α ⩽ string ]

1 + (match I with I → 42 | S → "quarante-deux")
```

Le typage de ce programme est très similaire à celui du chapitre 3. Le typage de l'application de l'opérateur `(+)` est très standard, il impose de typer :

$$\texttt{(match I with I} \rightarrow \texttt{42 | S} \rightarrow \texttt{"quarante-deux")} : \alpha_0$$

dans un ensemble de contraintes contenant $(\alpha_0 \leqslant \texttt{int})$. Ceci entraîne le typage de :

$$\texttt{I} : \alpha_1$$

dans un ensemble de contraintes contenant $(\alpha_1 \leqslant \alpha_2\ \texttt{t})$. Nous obtenons alors les contraintes $(\alpha_3\ \texttt{t} \leqslant \alpha_1)$ et $(\alpha_3 \leqslant \texttt{int})$ en utilisant la contrainte imposée sur le constructeur `I` lors de sa déclaration. Après saturation, nous obtenons en particulier la nouvelle contrainte $(\alpha_2 \leqslant \texttt{int})$.

Le typage du premier cas de filtrage génère la contrainte $(\alpha_2 \not\leqslant \texttt{int} \vee \texttt{int} \leqslant \alpha_0)$ qui entre en collision avec $(\alpha_2 \leqslant \texttt{int})$ et génère la contrainte $(\texttt{int} \leqslant \alpha_0)$. Par ailleurs, le typage du second cas de filtrage génère la contrainte $(\alpha_2 \not\leqslant \texttt{string} \vee \texttt{string} \leqslant \alpha_0)$ qui entre en collision avec $(\alpha_0 \leqslant \texttt{int})$ et génère la contrainte $(\alpha_2 \not\leqslant \texttt{string})$. Au final, la saturation se termine sans clash.

En revanche, si les deux constructeurs `I` et `S` sont permutés, le typage du code correspondant :

$$\texttt{1 + (match I with S} \rightarrow \texttt{42 | I} \rightarrow \texttt{"quarante-deux")} : \alpha_0$$

provoque la génération de la disjonction $(\alpha_2 \not\leqslant \texttt{int} \vee \texttt{string} \leqslant \alpha_0)$ dont le premier membre entre en collision avec $(\alpha_2 \leqslant \texttt{int})$ et le second avec $(\alpha_0 \leqslant \texttt{int})$, ce qui engendre correctement un clash.

Grâce à cette astuce, nous pouvons préserver l'inférence lors du typage du filtrage de motif sur des **`GADT`** et ne nécessitons pas d'annotation de type. Une annotation de type n'est nécessaire dans notre système uniquement pour la gestion de la récursion polymorphe. Ce problème sera traité spécifiquement dans la section 5.6.

5.2.4 Adaptation des mécanismes de saturation

Les contraintes que doivent maintenant gérer les règles de saturation contiennent des disjonctions et des négations. Les disjonctions sont gérées dans ce système exactement comme dans le chapitre 3. Le principe consiste à propager un ensemble Ψ de « contraintes alternatives » et, en cas de clash, lorsque le Ψ n'est pas vide, de rabattre la saturation sur l'un des membres du Ψ.

La négation d'une relation de sous-typage, notée $(\not\leqslant)$, est une notion nouvelle dans ce chapitre. La saturation des contraintes gère cette nouvelle relation de manière similaire à la gestion de la transitivité de la relation $(\leqslant)$ via quelques « lois » simples. Il est facile d'intuiter ces lois par de petits raisonnements sur les ensembles dénotés par les types. Par exemple :

$$A \subset B \ \wedge\ A \not\subset C \ \Rightarrow\ B \not\subset C$$

En effet, si trois ensembles A, B et C vérifient $A \subset B$ et $A \not\subset C$, alors il existe un $x \in A$ tel que $x \notin C$. Puisque $A \subset B$ et $x \in A$ alors $x \in B$. En conclusion $B \not\subset C$.

Cette propriété se traduit en particulier par la règle de saturation :

SNotLeqLeft

$$\frac{\begin{array}{c}\texttt{let}\ (\Psi_1, \tau_1^r), \ldots, (\Psi_p, \tau_p^r) = \textsc{rights}(\alpha,\ \Phi_1) \\ \Phi_1 \vdash \Psi \vee \Psi_1 \vee \tau_1^r \not\leqslant (\alpha_1, \ldots, \alpha_n)\ \mathtt{t} \rhd \Phi_2 \qquad \ldots \qquad \Phi_p \vdash \Psi \vee \Psi_p \vee \tau_p^r \not\leqslant (\alpha_1, \ldots, \alpha_n)\ \mathtt{t} \rhd \Phi_{p+1}\end{array}}{\Phi_1 \vdash \Psi \vee \alpha \not\leqslant (\alpha_1, \ldots, \alpha_n)\ \mathtt{t} \rhd \Phi_{p+1}}$$

signifiant que lors de la rencontre d'une nouvelle contrainte de la forme $(\Psi \vee \alpha \not\leqslant \tau^l)$, il faut extraire de l'ensemble de contraintes Φ toutes les disjonctions de la forme $(\Psi_i \vee \alpha \leqslant \tau_i^r)$ et tenter de prouver la validité et la compatibilité avec Φ des contraintes $(\Psi \vee \Psi_i \vee \tau_i^r \not\leqslant \tau^l)$.

La propriété duale :

$$A \subset B \ \wedge\ C \not\subset B \ \Rightarrow\ C \not\subset A$$

est gérée de manière similaire par (entre autres) la règle :

SNotLeqRight

$$\frac{\begin{array}{c}\texttt{let}\ (\Psi_1, \tau_1^l), \ldots, (\Psi_p, \tau_p^l) = \textsc{lefts}(\alpha,\ \Phi_1) \\ \Phi_1 \vdash \Psi \vee \Psi_1 \vee (\alpha_1, \ldots, \alpha_n)\ \mathtt{t} \not\leqslant \tau_1^l \rhd \Phi_2 \qquad \ldots \qquad \Phi_n \vdash \Psi \vee \Psi_p \vee (\alpha_1, \ldots, \alpha_n)\ \mathtt{t} \not\leqslant \tau_p^l \rhd \Phi_{p+1}\end{array}}{\Phi_1 \vdash \Psi \vee (\alpha_1, \ldots, \alpha_n)\ \mathtt{t} \not\leqslant \alpha \rhd \Phi_{p+1}}$$

Les règles de saturation concernant l'ajout de relations de la forme $(\alpha \leqslant \tau^r)$ doivent elles aussi être adaptées pour prendre en compte la potentielle présence de contraintes de la forme $(\alpha \not\leqslant \tau^l)$ dans l'ensemble de contraintes.

Les autres « lois » gérant la relation $(\not\leqslant)$ sont très intuitives. La liaison par $(\not\leqslant)$ entre deux types paramétrés de noms différents est toujours valide, ce qui se traduit par une règle de saturation sans prémisse. La liaison par $(\not\leqslant)$ entre deux types paramétrés de même nom est convertie en une disjonction de $(\not\leqslant)$ sur les paramètres des types. S'il n'y a aucun paramètre (l'élément neutre de l'opérateur $(\vee)$ étant `false`), il faut en quelque sorte « backtracker » sur l'ensemble de contraintes alternatives Ψ lorsqu'il n'est pas vide, et provoquer un clash dans le cas contraire.

5.2.5 Les types existentiels

L'exemple de code étudié dans la section 5.2.3 était volontairement simpliste. En effet, le **`GADT`** manipulé ne faisait intervenir que des variables de type apparaissant dans les paramètres, et les constructeurs de données `I` et `S` n'avaient en particulier pas de paramètre.

Comme nous l'avons expliqué précédemment, lors de la déclaration d'un nouveau type **`GADT`**, toutes les variables de type n'apparaissant pas en paramètre sont gérées de manière particulière et provoquent en particulier la génération de « types existentiels » lors du typage d'un filtrage.

Dans les systèmes de types classiques à base d'unification, les types existentiels sont habituellement gérés de manière similaire aux types abstraits : toute tentative d'unification entre un type existentiel et un type construit entraîne un clash.

Dans notre système, les types existentiels ne sont pas toujours « abstraits » et peuvent être liés entre eux, aux paramètres du type, ainsi qu'à d'autres types construits via les contraintes posées par l'utilisateur lors de la déclaration du **GADT**. Ces contraintes sont simplement « vérifiées » lors de la construction d'une valeur. Toute la difficulté consiste à utiliser correctement ces contraintes lors de la « déconstruction » d'une valeur via un filtrage de motifs.

Dans les exemples informels qui suivent, nous représentons les types existentiels de manière « classique » comme s'il s'agissait de simples identifiants uniques que nous notons « ϵ_1 », « ϵ_2 », ... Leur définition formelle, que nous donnerons par la suite, sera toutefois assez différente car les types existentiels « transporteront » un ensemble d'informations utiles pour la saturation des contraintes.

L'astuce de base est simple : étant donné un ensemble de contraintes connues Φ (provenant typiquement de la déclaration d'un **GADT**) mentionnant un type existentiel ϵ, pour vérifier la validité d'une contrainte de sous-typage `C` reliant ϵ à un autre type (typiquement générée lors du typage d'un cas de filtrage), nous allons chercher un ensemble de contraintes Φ' de telle sorte que Φ' et Φ impliquent `C`. Si un tel ensemble de contraintes Φ' existe, nous allons alors tenter de le prouver à la place de `C`.

Pour donner l'intuition des mécanismes de typage sous-jacents, observons le fonctionnement de cette technique sur quelques cas dégénérés :

1. Considérons l'exemple suivant :

   ```
   type t = A β [ β ⩽ int ]
   match A 123 with A n → n + 1
   ```

   Ici, le typage de `(A 123 :` α_0`)` impose `(t` $\leqslant \alpha_0$`)` et provoque la génération d'une variable de type classique α_1 devant vérifier ($\alpha_1 \leqslant$ `int`) à cause de la contrainte imposée lors de la définition du constructeur `A`. Le typage de `(123 :` α_1`)` génère la contrainte (`int` $\leqslant \alpha_1$) qui est bien compatible avec ($\alpha_1 \leqslant$ `int`).

   Par ailleurs, le typage du cas de filtrage génère un type existentiel ϵ_0 associé à la variable `n`. Le typage de l'expression `n + 1` impose de typer `(n :` α_2`)` avec ($\alpha_2 \leqslant$ `int`) ce qui engendre, après saturation, la contrainte ($\epsilon_0 \leqslant$ `int`).

   Grâce à la contrainte ($\beta \leqslant$ `int`) imposée sur le constructeur de données `A`, nous reportons la contrainte $\epsilon_0 \leqslant$ `int` sur `int` et obtenons (`int` $\leqslant$ `int`).

   La contrainte que nous avons obtenue ici (à savoir ($\epsilon_0 \leqslant$ `int`)) est une instance directe de celle mentionnée lors de la déclaration du constructeur (à savoir ($\beta \leqslant$ `int`)). Un simple test d'appartenance aurait suffit ici, mais ce n'est pas toujours le cas comme nous allons le voir maintenant.

2. Considérons l'exemple suivant :

   ```
   type α t = B β [ β ⩽ α ]
   match B 3 with B n → n + 1
   ```

Le typage de `(B 3 :` α_0`)` provoque la génération de deux variables de type classiques α_1 et α_2 vérifiant $(\alpha_1$ `t` $\leqslant \alpha_0)$ et $(\alpha_2 \leqslant \alpha_1)$. Le typage de `(3 :` α_2`)` impose alors $($`int` $\leqslant \alpha_2)$ et donc $($`int` $\leqslant \alpha_1)$ par transitivité.

Le typage du filtrage s'effectuant sur un constructeur de type `t`, il génère une variable classique α_3 vérifiant $(\alpha_0 \leqslant \alpha_3$ `t`$)$ et un type existentiel ϵ_0 associé à la variable `n` du programme. Le typage de l'expression `n + 1` impose alors de typer `(n :` α_4`)` avec $(\alpha_4 \leqslant$ `int`$)$ ce qui engendre, après saturation, la contrainte $(\epsilon_0 \leqslant$ `int`$)$.

Pour saturer cette dernière contrainte, comme précédemment, nous utilisons la seule contrainte connue sur ϵ_0 (à savoir $(\epsilon_0 \leqslant \alpha_3)$) provenant de la définition du constructeur `B` et construisons la contrainte $(\alpha_3 \leqslant$ `int`$)$. En effet, la règle standard sur la transitivité nous montre que $(\epsilon_0 \leqslant \alpha_3)$ et $(\alpha_3 \leqslant$ `int`$)$ impliquent bien $(\epsilon_0 \leqslant$ `int`$)$. Pour montrer que la contrainte $(\epsilon_0 \leqslant$ `int`$)$ ne casse pas la validité du typage, la seule solution est de démontrer que $(\alpha_3 \leqslant$ `int`$)$ est compatible avec les autres contraintes. La contrainte sur le type existentiel a donc dans ce cas été repoussée à l'identique sur l'argument du type.

La saturation se termine alors comme habituellement, et le `int` généré lors du typage de `3` est correctement comparé au `int` généré lors du typage de l'application du `(+)` en suivant la chaîne $($`int` $\leqslant \alpha_2 \leqslant \alpha_1 \leqslant \alpha_3 \leqslant$ `int`$)$.

3. Revenons maintenant sur une variante plus complète de l'exemple consistant à retarder un appel de fonction :

```
type α box = Box (β1, β2) [ β1 ⩽ β2 → α ]
match Box (string_of_int, 42) with
| Box (f, x) → f x
```

où `string_of_int` est une fonction de conversion d'un entier en une chaîne de caractères dont le schéma de type est :

$$\texttt{string\_of\_int} : [\, \forall\alpha_1\alpha_2\alpha_3 \,.\, \alpha_1 \mid \alpha_2 \to \alpha_3 \leqslant \alpha_1 \ \wedge\ \alpha_2 \leqslant \texttt{int} \ \wedge\ \texttt{string} \leqslant \alpha_3 \,]$$

Puisque la définition du constructeur de données `Box` mentionne trois variables de type (α, β_1 et β_2), le typage de `(Box (string_of_int, 42) :` α_4`)` provoque la génération de trois variables de type fraîches, que nous nommerons respectivement α_5, α_6 et α_7 et vérifiant $(\alpha_5$ `box` $\leqslant \alpha_4)$. L'instanciation de la contrainte provenant de la déclaration du constructeur `Box` est $(\alpha_6 \leqslant \alpha_7 \to \alpha_5)$. Nous typons alors `(string_of_int :` α_6`)` et `(42 :` α_7`)`. La saturation des contraintes se termine correctement et impose en particulier $($`string` $\leqslant \alpha_5)$.

Par ailleurs, le typage du cas de filtrage génère une variable de type universelle α_8 (instance du α de la définition du type `box`) et deux types existentiels ϵ_0 et ϵ_1 (correspondant aux β_1 et β_2 de cette même définition) vérifiant $(\alpha_4 \leqslant \alpha_8$ `box`$)$. Après saturation, nous obtenons en particulier $(\alpha_5 \leqslant \alpha_8)$.

Le typage de l'appel `(f x :` α_9`)` se fait alors dans l'environnement de typage contenant les deux associations `(f :` ϵ_0`)` et `(x :` ϵ_1`)`. Il provoque la génération de trois variables de type α_{10}, α_{11} et α_{12} vérifiant les contraintes $(\alpha_{10} \leqslant \alpha_{11} \to \alpha_{12})$, $(\alpha_{12} \leqslant \alpha_9)$, $(\epsilon_0 \leqslant \alpha_{10})$ et $(\epsilon_1 \leqslant \alpha_{11})$.

L'algorithme de saturation engendre en particulier la contrainte $(\epsilon_0 \leqslant \alpha_{11} \to \alpha_{12})$. Il s'agit d'une comparaison entre un type existentiel et un type paramétré $(\to)$. La contrainte provenant de la déclaration du constructeur `Box` permet de saturer cette comparaison en engendrant les contraintes $(\alpha_{13} \to \alpha_8 \leqslant \alpha_{11} \to \alpha_{12})$, $(\alpha_{13} \leqslant \epsilon_1)$ et $(\epsilon_1 \leqslant \alpha_{13})$. La variable intermédiaire α_{13} permet ici de conserver les types sous forme normale (pour rappel, la forme normale impose en particulier qu'un constructeur de type ne peut être paramétré que par des variables de type universelles).

Après saturation, nous obtenons bien $(\texttt{string} \leqslant \alpha_5 \leqslant \alpha_8 \leqslant \alpha_{12} \leqslant \alpha_9)$ indiquant que le résultat du calcul est une chaîne de caractères ; et $(\epsilon_1 \leqslant \alpha_{11} \leqslant \alpha_{13} \leqslant \epsilon_1)$ correspondant à la vérification de la compatibilité entre le type de l'argument et le type du paramètre de la fonction lors de l'appel.

De manière générale, les contraintes annotant les constructeurs de données lors de leur déclaration ne sont pas forcément réduites à une conjonction de contraintes faisant intervenir une seule fois chaque type existentiel à gauche et/ou à droite d'une relation de sous-typage.

Il est possible d'étendre manuellement la technique décrite ci-dessus à plus de cas. En particulier, lorsque l'annotation du constructeur est de la forme « $\cdots \wedge \beta \leqslant \tau_1 \wedge \cdots \wedge \beta \leqslant \tau_2 \wedge \cdots \wedge \beta \leqslant \tau_3 \wedge \ldots$ », c'est-à-dire lorsqu'il consiste une conjonction faisant intervenir plusieurs comparaisons dans le même sens sur un même type existentiel, pour saturer une contrainte de la forme $\epsilon \leqslant \tau^r$ avec ϵ une instance de β, la contrainte la plus générale que nous pouvons générer pour remplacer $(\epsilon \leqslant \tau^r)$ est la disjonction $(\tau_1 \leqslant \tau^r \vee \tau_2 \leqslant \tau^r \vee \tau_3 \leqslant \tau^r)$.

Nous définissons formellement dans ce chapitre un mécanisme général permettant, à partir d'un ensemble de contraintes Φ quelconque provenant de la déclaration d'un constructeur et d'une contrainte de sous-typage mettant en relation un type existentiel et un type construit, le nouvel ensemble de contraintes le plus général devant être prouvé.

5.2.6 Portée des variables de type

Malheureusement, bloquer l'unification entre tout type existentiel et tout type construit, ou reporter toute comparaison entre un type existentiel et un type construit comme décrit précédemment, ne suffit pas pour assurer la validité du typage. Une seconde « vérification » doit être effectuée : les types existentiels ne doivent pas « s'échapper de leur portée ».

Pour mettre en évidence cette notion, commençons par analyser le code suivant :

```
(* Définition du même type box que précédemment.            *)
type box = Box (β → unit, β)

(* Définition d'une référence stockant une fonction.        *)
let rf = ref (fun _ → ())

(* Ouverture d'une boîte, appel (potentiellement invalide)  *)
(* de la fonction contenue dans la référence et remplacement*)
(* de son contenu par fonction provenant de la boîte.       *)
let precall box = match box with Box (f, x) → !rf x; rf := f

(* Deux appels à precall avec des boîtes de contenu         *)
(* incompatibles.                                           *)
precall (Box (print_string, "hello"));
precall (Box (print_int, 42));
```

L'exécution d'un tel code part en erreur puisqu'elle provoque un appel de la fonction `print_string` avec l'argument 42. Par conséquent, tout système de types valide doit refuser ce code, ce qui n'est malheureusement pas si évident. En effet, ce code contient un unique point de déconstruction d'une « `Box` », et il n'y a aucune raison particulière d'unifier ou de comparer le type existentiel généré en ce point avec un type construit. La seule comparaison ou unification potentielle qu'est susceptible de subir ce type existentiel est avec la variable de type (classique) générée lors du typage du `(fun _ → ())` pour le paramètre de la fonction.

Plus précisément, le problème vient du fait qu'un même point de déconstruction est exécuté plusieurs fois, et que les multiples exécutions de ce même cas de filtrage utilisent la référence (externe au filtrage) pour partager des valeurs.

Pour interdire un tel comportement invalide dans un système à base d'unification, une technique standard consiste à vérifier qu'à la sortie du typage d'un cas de filtrage, aucun type existentiel généré lors du typage du cas de filtrage en question n'est accessible via l'environnement de typage. Une telle vérification assure en effet qu'aucune unification ne s'est produite lors du typage d'un cas de filtrage entre un type existentiel local et une variable de type externe.

Une variante classique de cette technique consiste à associer aux variables de type un « niveau » représentant la profondeur du noeud de l'expression qui a provoqué leur génération. Il suffit alors d'interdire l'unification entre un type existentiel et une variable de type de niveau inférieur.

Comme nous le verrons par la suite, dans notre système, nous utilisons une variante de cette dernière méthode en conservant, pour chaque type existentiel, un intervalle de variables de type (correspondant à celles générées lors du typage du cas de filtrage en question) avec lesquelles elles peuvent être comparées librement. La comparaison d'un type existentiel avec une variable

de type externe à cet intervalle n'est pas interdite, elle est simplement gérée exactement comme une comparaison avec un type construit.

5.3 Définition du système de types

Nous allons maintenant définir formellement le système de types de ce chapitre. Nous commençons par définir les notations que nous allons utiliser et la forme des règles d'inférence.

5.3.1 Notations utilisées

L'ensemble des types que nous allons manipuler est défini par la grammaire suivante :

$$\begin{array}{lll} \tau^l & \texttt{::=} & \alpha \texttt{ | (}\alpha_1\texttt{, ..., }\alpha_n\texttt{) t | } \epsilon(\alpha, \Phi, [\alpha^l; \alpha^r], \{\alpha_1, \ldots, \alpha_m\}) \\ \tau^r & \texttt{::=} & \tau^l \end{array}$$

Il n'y a pas de distinction entre τ^r et τ^l dans ce système. Les lois sur la conservation de la position des τ^r et τ^l dans les contraintes manipulées lors du typage et de la saturation restent néanmoins valides.

Les types existentiels que nous définissons ici sont structurellement assez différents des ceux définis dans les systèmes classiques. Il ne s'agit pas ici de simples « identifiants uniques » comme les variables de type universelles, mais d'une structure de la forme $\epsilon(\alpha, \Phi, [\alpha^l; \alpha^r], \{\alpha_1, \ldots, \alpha_m\})$ dans laquelle :

- « ϵ » est un simple élément de syntaxe spécifiant qu'il s'agit d'un type existentiel, au même titre que « (», «) », « , », « [», et «] ».
- « α » est une variable de type universelle pouvant apparaître dans « Φ ». Il s'agit obligatoirement, par construction, de l'un des $\alpha_1, \ldots, \alpha_m$.
- « Φ » est l'ensemble de contraintes provenant de la déclaration du constructeur du **`GADT`** correspondant dans lequel les variables de types ont été instanciées.
- « α^l » et « α^r » sont deux variables de type universelles représentant la « portée » du type existentiel. Ils permettent de tester si une variable universelle α' est dans ou hors de cette portée. Un tel test sera noté « $\alpha' \in [\alpha^l; \alpha^r]$ ».
- « $\alpha_1, \ldots, \alpha_m$ » sont m variables de type universelles. Il s'agit des instances des variables de type présentes dans la définition du constructeur du **`GADT`** correspondant, qui ne sont pas des paramètres du **`GADT`**.

Les types existentiels sont créés lors du typage du filtrage de motifs et utilisés lors de la saturation des contraintes.

Les ensembles de contraintes (Φ) sous forme normale conjonctive et (Ψ) sous forme normale disjonctive sont définis par la grammaire suivante :

$$\begin{array}{lcl} \mathtt{C} & ::= & \tau^l \leqslant \tau^r \mid \tau^r \nleqslant \tau^l \\ \psi & ::= & \mathtt{C}_1 \vee \ldots \vee \mathtt{C}_n \\ \Phi & ::= & \psi_1 \wedge \ldots \wedge \psi_n \\ \phi & ::= & \mathtt{C}_1 \wedge \ldots \wedge \mathtt{C}_n \\ \Psi & ::= & \phi_1 \vee \ldots \vee \phi_n \end{array}$$

La définition d'un nouveau **GADT** est de la forme suivante :

```
type (α1, ..., αn) t =
  | ∃ β1, ..., βp1 . K1 (β1, ..., βq1) [ Φ1 ]
  | ...
  | ∃ β1, ..., βpr . Kr (β1, ..., βqr) [ Φr ]
```

avec $n \geqslant 0$ et $r \geqslant 1$; et pour tout $i \in [\,1;\ r]$, $0 \leqslant q_i \leqslant p_i$. Pour simplifier les règles de typage, nous imposons les variables qui ne sont pas des paramètres du type (c'est-à-dire différentes de $\alpha_1, \ldots, \alpha_n$) mentionnées en argument des constructeurs de données et dans les ensemble de contraintes $\Phi_1, \ldots, \Phi_r$ a être déclarées via un $\exists$.

Par ailleurs, les ensembles de contraintes $\Phi_1, \ldots, \Phi_r$ doivent tous être valides et saturés. Nous remarquerons que la validité de ces Φ_i n'est pas obligatoire pour la validité du système de types. Néanmoins, si l'un des Φ_i contenait une contrainte invalide, toute utilisation, dans le code, du constructeur de données K_i pour créer une valeur de type $(\alpha_1, \ldots, \alpha_n)$ `t` provoquerait un clash de typage. La saturation des Φ_i n'est pas non plus nécessaire pour la validité du système, mais elle renforce la puissance du système de types dans sa gestion du filtrage de motifs sans réduire sa puissance dans sa gestion de la construction de valeurs de type $(\alpha_1, \ldots, \alpha_n)$ `t`. En pratique, il n'est pas nécessaire d'imposer au programmeur de saturer manuellement ces ensembles de contraintes, le typeur le fait à la volée, tout en vérifiant leur validité.

Les schémas de type ont une structure similaire à ceux des chapitres précédents, la seule différence tient au fait que les ensembles de contraintes qu'ils contiennent sont maintenant plus « riches » :

$$\sigma \quad ::= \quad [\,\forall \alpha_1 \ldots \alpha_n \,.\, \alpha \mid \Phi\,]$$

Comme précédemment, nous définissons quelques fonctions utilitaires permettant d'extraire d'un ensemble de contraintes des relations concernant une variable de type donnée :

$$\begin{array}{rcl} \textsc{lefts}(\alpha,\ \Phi) & \triangleq & \{\ (\Psi, \tau^l) \mid (\Psi \vee \tau^l \leqslant \alpha) \in \Phi\ \} \\ \textsc{rights}(\alpha,\ \Phi) & \triangleq & \{\ (\Psi, \tau^r) \mid (\Psi \vee \alpha \leqslant \tau^r) \in \Phi\ \} \\ \overline{\textsc{lefts}}(\alpha,\ \Phi) & \triangleq & \{\ (\Psi, \tau^r) \mid (\Psi \vee \tau^r \nleqslant \alpha) \in \Phi\ \} \\ \overline{\textsc{rights}}(\alpha,\ \Phi) & \triangleq & \{\ (\Psi, \tau^l) \mid (\Psi \vee \alpha \nleqslant \tau^l) \in \Phi\ \} \end{array}$$

Nous définissons en plus les quatre fonctions suivantes, similaires aux précédentes. Elle prennent un argument supplémentaire composé d'un ensemble de variables de type universelles ne devant pas apparaître dans le résultat.

$$\begin{aligned}
\not\in\textsc{lefts}(\alpha,\ \Phi,\ \{\alpha_1,\ldots,\alpha_m\}) &\triangleq \{\ (\Psi,\tau^l)\mid(\Psi\vee\tau^l\leqslant\alpha)\in\Phi\ \wedge\ \tau^l\notin\{\alpha_1,\ldots,\alpha_m\}\ \}\\
\not\in\textsc{rights}(\alpha,\ \Phi,\ \{\alpha_1,\ldots,\alpha_m\}) &\triangleq \{\ (\Psi,\tau^r)\mid(\Psi\vee\alpha\leqslant\tau^r)\in\Phi\ \wedge\ \tau^r\notin\{\alpha_1,\ldots,\alpha_m\}\ \}\\
\overline{\not\in\textsc{lefts}}(\alpha,\ \Phi,\ \{\alpha_1,\ldots,\alpha_m\}) &\triangleq \{\ (\Psi,\tau^r)\mid(\Psi\vee\tau^r\not\leqslant\alpha)\in\Phi\ \wedge\ \tau^r\notin\{\alpha_1,\ldots,\alpha_m\}\ \}\\
\overline{\not\in\textsc{rights}}(\alpha,\ \Phi,\ \{\alpha_1,\ldots,\alpha_m\}) &\triangleq \{\ (\Psi,\tau^l)\mid(\Psi\vee\alpha\not\leqslant\tau^l)\in\Phi\ \wedge\ \tau^l\notin\{\alpha_1,\ldots,\alpha_m\}\ \}
\end{aligned}$$

Ces fonctions sont utilisées dans certaines règles de saturation faisant intervenir un type existentiel. L'ensemble de variables universelles passé en troisième argument est alors le dernier membre du type existentiel en question.

Comme précédemment, dans ces huit définitions, les disjonctions peuvent être considérées à permutation près ou pas, sans que cela change l'ensemble des expressions acceptées par le système de types. Ce choix sera discuté dans le chapitre 6 sur l'implémentation car son impact sur les performances est très élevé.

5.3.2 Forme des règles d'inférence

Comme dans les systèmes des chapitres précédents, les règles d'inférence peuvent être classifiées dans trois catégories :

- Les règles de typage, de la forme :

$$\frac{\mathrm{T}\ldots\quad\ldots\quad\ldots\quad\ldots}{\Phi,\Gamma\vdash\Psi\vee\mathtt{e}:\alpha\rhd\Phi'}\qquad\qquad\frac{\mathrm{T}\ldots\quad\ldots\quad\ldots\quad\ldots}{\Phi,\Gamma\vdash\Psi\vee\mathtt{K}(\mathtt{x}_1,\ldots,\mathtt{x}_p)\to\mathtt{e}:\alpha_e,\alpha'_1,\ldots\alpha'_n\rhd\Phi'}$$

$$\frac{\mathrm{T}\ldots\quad\ldots\quad\ldots\quad\ldots}{\Phi\vdash\Psi\vee\mathtt{K}\to\mathtt{CLASH}:\alpha'_1,\ldots\alpha'_n\rhd\Phi'}$$

  Ces règles ont en prémisse des conclusions de règles de typage, d'instanciation et de saturation. La première forme est standard. Les deux autres sont utilisées pour exprimer le typage du filtrage de motif. La dernière permet en particulier de gérer les cas absents dans un filtrage partiel.
- La règle d'instanciation, de la forme :

$$\frac{\mathrm{I}\ldots\quad\ldots\quad\ldots\quad\ldots}{\Phi\vdash\Psi\vee\sigma\leqslant\tau^r\rhd\Phi'}$$

Tout comme dans le système de base et le système du chapitre 3 (mais pas le chapitre 4), cette règle est utilisée à la limite entre la partie typage de l'arbre d'inférence et la partie saturation. Elle possède en prémisse uniquement des conclusions de règles de saturation.

- Les règles de saturation, de la forme :

$$\frac{\mathsf{S}\ldots \quad \cdots \quad \cdots \quad \cdots}{\Phi \vdash \Psi \vee (\mathsf{C}_1 \wedge \cdots \wedge \mathsf{C}_n) \rhd \Phi'}$$

$$\frac{\mathsf{S}\ldots \quad \cdots \quad \cdots \quad \cdots}{\Phi \vdash \Psi \vee \tau^l \leqslant \tau^r \rhd \Phi'} \qquad \frac{\mathsf{S}\ldots \quad \cdots \quad \cdots \quad \cdots}{\Phi \vdash \Psi \vee \tau^l \leq \tau^r \rhd \Phi'}$$

$$\frac{\mathsf{S}\ldots \quad \cdots \quad \cdots \quad \cdots}{\Phi \vdash \Psi \vee \tau^r \nleqslant \tau^l \rhd \Phi'} \qquad \frac{\mathsf{S}\ldots \quad \cdots \quad \cdots \quad \cdots}{\Phi \vdash \Psi \vee \tau^r \nleq \tau^l \rhd \Phi'}$$

La première forme est utilisée pour traiter la présence de conjonctions dans Ψ. Après « éclatement » des conjonctions, les membres des disjonctions manipulées sont de la forme $\tau^l \leqslant \tau^r$ et $\tau^r \nleqslant \tau^l$. La saturation des contraintes consiste comme habituellement à calculer un point fixe sur Φ. Lorsqu'une contrainte de la forme $(\tau^l \leqslant \tau^r)$ (respectivement $(\tau^r \nleqslant \tau^l)$) n'a pas encore été ajoutée à Φ, elle y est ajoutée et la contrainte correspondante $\tau^l \leq \tau^r$ (respectivement $(\tau^r \nleq \tau^l)$) est générée.

Comme dans le chapitre 3, « typer » une expression e consiste à générer une variable de type fraîche α et tenter de construire l'arbre d'inférence au dessus de :

$$\frac{\bigtriangledown}{\varnothing, \varnothing \vdash \varnothing \vee \mathsf{e} : \alpha \rhd \Phi}$$

Si l'arbre d'inférence n'est pas constructible, l'expression est dite « non typable ». Sinon, le schéma de type inféré pour e est $\textsc{gen}(\alpha, \Phi, \varnothing)$ où la fonction $\textsc{gen}$ peut être définie naïvement comme :

$$\textsc{gen}(\alpha, \Phi, \Gamma) \triangleq [\ \forall(\textsc{ftv}(\Phi) \backslash \textsc{ftv}(\Gamma))\ .\ \alpha \mid \Phi\ ]$$

où $\textsc{ftv}(X)$ représente l'ensemble des variables de type universelles libres de X. Cette définition de $\textsc{gen}$ peut être « améliorée » pour générer un schéma de type équivalent, mais nettoyé et donc plus « lisible » comme nous le verrons dans le prochain chapitre sur l'implémentation.

5.3.3 Règles de typage

Les règles de typage sont très similaires à celles du chapitre 3 :

- Le typage des constantes se fait via la règle suivante :

$$\textsc{TConst}\ \frac{\Phi \vdash \Psi \vee T(\mathtt{c}) \leqslant \alpha \rhd \Phi'}{\Phi, \Gamma \vdash \Psi \vee \mathtt{c} : \alpha \rhd \Phi'}$$

  La méta-fonction T génère une schéma de type. Toute utilisation de cette règle engendre donc une utilisation immédiate de la règle d'instanciation (Inst).

- Le typage de l'application d'une primitive se fait par l'une des règles suivantes en fonction de son arité :

$$\textsc{TApplyPrim1}\ \frac{\mathtt{let}\ \alpha_1, \alpha_2\ \textit{fresh} \qquad \Phi \vdash \Psi \vee T(\mathtt{p}^1) \leqslant \alpha_1 \to \alpha_2 \rhd \Phi' \qquad \Phi' \vdash \Psi \vee \alpha_2 \leqslant \alpha \rhd \Phi'' \qquad \Phi'', \Gamma \vdash \Psi \vee \mathtt{e}_1 : \alpha_1 \rhd \Phi'''}{\Phi, \Gamma \vdash \Psi \vee \mathtt{p}^1\ \mathtt{e}_1 : \alpha \rhd \Phi'''}$$

$$\textsc{TApplyPrim2}\ \frac{\begin{array}{c}\mathtt{let}\ \alpha_0, \alpha_1, \alpha_2, \alpha_3\ \textit{fresh} \qquad \Phi \vdash \Psi \vee \alpha_0 \leqslant \alpha_2 \to \alpha_3 \rhd \Phi' \\ \Phi' \vdash \Psi \vee \alpha_3 \leqslant \alpha \rhd \Phi'' \qquad \Phi'' \vdash \Psi \vee T(\mathtt{p}^2) \leqslant \alpha_1 \to \alpha_0 \rhd \Phi''' \\ \Phi''', \Gamma \vdash \Psi \vee \mathtt{e}_1 : \alpha_1 \rhd \Phi'''' \qquad \Phi'''', \Gamma \vdash \Psi \vee \mathtt{e}_2 : \alpha_2 \rhd \Phi'''''\end{array}}{\Phi, \Gamma \vdash \Psi \vee \mathtt{p}^2\ \mathtt{e}_1\ \mathtt{e}_2 : \alpha \rhd \Phi'''''}$$

- De la même manière, le typage des variables se fait par :

$$\textsc{TVar}\ \frac{\mathtt{when}\ \Gamma[\mathtt{x}]\ \textit{defined} \qquad \Phi \vdash \Psi \vee \Gamma[\mathtt{x}] \leqslant \alpha \rhd \Phi'}{\Phi, \Gamma \vdash \Psi \vee \mathtt{x} : \alpha \rhd \Phi'}$$

- Le typage des définitions et applications de fonctions est géré ainsi :

$$\textsc{TLambda}\ \frac{\mathtt{let}\ \alpha_1, \alpha_2\ \textit{fresh} \qquad \Phi, \Gamma \oplus (\mathtt{x}, \alpha_1) \vdash \Psi \vee \mathtt{e} : \alpha_2 \rhd \Phi' \qquad \Phi' \vdash \Psi \vee \alpha_1 \to \alpha_2 \leqslant \alpha \rhd \Phi''}{\Phi, \Gamma \vdash \Psi \vee \lambda\ \mathtt{x}\ .\ \mathtt{e} : \alpha \rhd \Phi''}$$

$$\textsc{TApp}\ \frac{\mathtt{let}\ \alpha_1, \alpha_2, \alpha_3\ \textit{fresh} \qquad \Phi \vdash \Psi \vee \alpha_1 \leqslant \alpha_2 \to \alpha_3 \rhd \Phi' \qquad \Phi' \vdash \Psi \vee \alpha_3 \leqslant \alpha \rhd \Phi'' \qquad \Phi'', \Gamma \vdash \Psi \vee \mathtt{e}_1 : \alpha_1 \rhd \Phi''' \qquad \Phi''', \Gamma \vdash \Psi \vee \mathtt{e}_2 : \alpha_2 \rhd \Phi''''}{\Phi, \Gamma \vdash \Psi \vee \mathtt{e}_1\ \mathtt{e}_2 : \alpha \rhd \Phi''''}$$

- Le typage des couples, de la conditionnelle et du `let` se fait encore une fois de la même manière que dans le chapitre 3 :

TPair

$$\frac{\begin{array}{c}\texttt{let}\ \alpha_1, \alpha_2\ \textit{fresh} \\ \Phi, \Gamma \vdash \Psi \vee \texttt{e}_1 : \alpha_1 \rhd \Phi' \qquad \Phi', \Gamma \vdash \Psi \vee \texttt{e}_2 : \alpha_2 \rhd \Phi'' \qquad \Phi'' \vdash \Psi \vee \alpha_1 \times \alpha_2 \leqslant \alpha \rhd \Phi'''\end{array}}{\Phi, \Gamma \vdash \Psi \vee (\texttt{e}_1,\ \texttt{e}_2) : \alpha \rhd \Phi'''}$$

TIf

$$\frac{\begin{array}{c}\texttt{let}\ \alpha'\ \textit{fresh} \qquad \Phi \vdash \Psi \vee \alpha' \leqslant \texttt{bool} \rhd \Phi' \\ \Phi', \Gamma \vdash \Psi \vee \texttt{e}_1 : \alpha' \rhd \Phi'' \qquad \Phi'', \Gamma \vdash \Psi \vee \texttt{e}_2 : \alpha \rhd \Phi''' \qquad \Phi''', \Gamma \vdash \Psi \vee \texttt{e}_3 : \alpha \rhd \Phi''''\end{array}}{\Phi, \Gamma \vdash \Psi \vee \texttt{if}\ \texttt{e}_1\ \texttt{then}\ \texttt{e}_2\ \texttt{else}\ \texttt{e}_3 : \alpha \rhd \Phi''''}$$

TLet

$$\frac{\texttt{let}\ \alpha'\ \textit{fresh} \qquad \Phi, \Gamma \vdash \Psi \vee \texttt{e}_1 : \alpha' \rhd \Phi' \qquad \Phi', \Gamma \oplus (\texttt{x}, \textsc{gen}(\alpha', \Phi', \Gamma)) \vdash \Psi \vee \texttt{e}_2 : \alpha \rhd \Phi''}{\Phi, \Gamma \vdash \Psi \vee \texttt{let}\ \texttt{x} = \texttt{e}_1\ \texttt{in}\ \texttt{e}_2 : \alpha \rhd \Phi''}$$

- La première différence fondamentale concerne l'utilisation d'un constructeur de données permettant de créer une valeur de type **`GADT`**. Dans cette situation, nous utilisons la définition du **`GADT`** en question et imposons les contraintes de sous-typage qu'elle mentionne en instanciant les différentes variables de type :

TConstr

$$\frac{\begin{array}{c}\texttt{when type}\ (\alpha_1, \ldots, \alpha_n)\ \texttt{t} = \ldots \mid \exists \beta_1, \ldots, \beta_p\ .\ \texttt{K}(\beta_1, \ldots, \beta_q)\ [\ \Phi_0\ ] \mid \ldots \\ \texttt{let}\ \alpha'_1, \ldots, \alpha'_n, \beta'_1, \ldots, \beta'_p\ \textit{fresh} \qquad \Phi \vdash \Psi \vee \Phi_0[\alpha_i \mapsto \alpha'_i]_{i=1}^{n}[\beta_i \mapsto \beta'_i]_{i=1}^{p} \rhd \Phi_1 \\ \Phi_1 \vdash \Psi \vee e_1 : \beta'_1 \rhd \Phi_2 \quad \ldots \quad \Phi_q \vdash \Psi \vee e_q : \beta'_q \rhd \Phi_{q+1} \\ \Phi_{q+1} \vdash \Psi \vee (\alpha'_1, \ldots, \alpha'_n)\ \texttt{t} \leqslant \alpha \rhd \Phi_{q+2}\end{array}}{\Phi, \Gamma \vdash \Psi \vee \texttt{K}\ (e_1, \ldots, e_q) : \alpha \rhd \Phi_{q+2}}$$

- La seconde différence concerne le typage du filtrage de motifs. Il est défini par les trois règles suivantes :

TMatch

$$\frac{\begin{array}{c}\texttt{when type}\ (\alpha_1, \ldots, \alpha_p)\ \texttt{t} = \{\ \exists \beta_{i,1}, \ldots, \beta_{i,q_i}\ .\ \texttt{K}_i(\beta_{i,1}, \ldots, \beta_{i,r_i})\ [\ \Phi_i\ ]\ \}_{i=1}^{s} \\ \texttt{let}\ \alpha_e, \alpha'_1, \ldots, \alpha'_p\ \textit{fresh} \\ \Phi, \Gamma \vdash \Psi \vee \texttt{e} : \alpha_e \rhd \Phi' \qquad \Phi', \Gamma \vdash \Psi \vee \alpha_e \leqslant (\alpha'_1, \ldots, \alpha'_p)\ \texttt{t} \rhd \Phi'' \\ \Phi'', \{\ \Gamma \vdash \Psi \vee \texttt{K}_i\ (\texttt{x}_{i,1}, \ldots, \texttt{x}_{i,r_i}) \to \texttt{e}_i : \alpha_e, \alpha'_1, \ldots, \alpha'_p\ \}_{i=1}^{n} \rhd \Phi''' \\ \Phi''' \vdash \{\ \Psi \vee \texttt{K}_i \to \texttt{CLASH} : \alpha'_1, \ldots, \alpha'_p\ \}_{i=n+1}^{s} \rhd \Phi''''\end{array}}{\Phi, \Gamma \vdash \Psi \vee \texttt{match e with}\ \texttt{K}_1\ (\texttt{x}_{1,1}, \ldots, \texttt{x}_{1,r_1}) \to \texttt{e}_1\ \|\ \ldots\ \|\ \texttt{K}_n\ (\texttt{x}_{n,1}, \ldots, \texttt{x}_{n,r_n}) \to \texttt{e}_n : \alpha \rhd \Phi''''}$$

TCase

$$\frac{\begin{array}{c}\texttt{when}\ \ \texttt{type}\ (\alpha_1,...,\alpha_n)\ \texttt{t} = ... \mid \exists\, \beta_1,...,\beta_q\,.\, K(\beta_1,...,\beta_p)\ [\Phi_0] \mid ... \\ \texttt{let}\ \ \alpha^l\ \textit{fresh} \qquad \texttt{let}\ \ \beta'_1,...,\beta'_q\ \textit{fresh} \qquad \texttt{let}\ \Phi_1 = \Phi_0[\alpha_i \mapsto \alpha'_i]_{i=1}^n[\beta_i \mapsto \beta'_i]_{i=1}^q \\ \Phi, \Gamma \oplus \{\ \texttt{x}_i : \beta'_i\ \}_{i=1}^p \vdash \Psi \vee \overline{\Phi_1} \vee \texttt{e} : \alpha_e \rhd \Phi' \qquad \texttt{let}\ \ \alpha^r\ \textit{fresh} \\ \Phi' \vdash \{\ \Psi \vee \epsilon(\beta'_i, \Phi_1, [\alpha^l; \alpha^r], \{\beta'_1, \ldots, \beta'_q\}) \leqslant \beta'_i\ \}_{i=1}^q \rhd \Phi'' \\ \Phi'' \vdash \{\ \Psi \vee \beta'_i \leqslant \epsilon(\beta'_i, \Phi_1, [\alpha^l; \alpha^r], \{\beta'_1, \ldots, \beta'_q\})\ \}_{i=1}^q \rhd \Phi'''\end{array}}{\Phi, \Gamma \vdash \Psi \vee \texttt{K}(\texttt{x}_1, ..., \texttt{x}_p) \to \texttt{e} : \alpha_e, \alpha'_1, ..., \alpha'_n \rhd \Phi'''}$$

TAbsentCase

$$\frac{\begin{array}{c}\texttt{when}\ \ \texttt{type}\ (\alpha_1,...,\alpha_n)\ \texttt{t} = ... \mid \exists\, \beta_1,...,\beta_q\,.\, K(\beta_1,...,\beta_p)\ [\Phi_0] \mid ... \\ \texttt{let}\ \ \alpha^l\ \textit{fresh} \qquad \texttt{let}\ \ \beta'_1,...,\beta'_q\ \textit{fresh} \qquad \texttt{let}\ \Phi_1 = \Phi_0[\alpha_i \mapsto \alpha'_i]_{i=1}^n[\beta_i \mapsto \beta'_i]_{i=1}^q \\ \Phi \vdash \Psi \vee \overline{\Phi_1} \rhd \Phi' \qquad \texttt{let}\ \ \alpha^r\ \textit{fresh} \\ \Phi' \vdash \{\ \Psi \vee \epsilon(\beta'_i, \Phi_1, [\alpha^l; \alpha^r], \{\beta'_1, \ldots, \beta'_q\}) \leqslant \beta'_i\ \}_{i=1}^q \rhd \Phi'' \\ \Phi'' \vdash \{\ \Psi \vee \beta'_i \leqslant \epsilon(\beta'_i, \Phi_1, [\alpha^l; \alpha^r], \{\beta'_1, \ldots, \beta'_q\})\ \}_{i=1}^q \rhd \Phi'''\end{array}}{\Phi \vdash \Psi \vee \texttt{K} \to \texttt{CLASH} : \alpha'_1, ..., \alpha'_n \rhd \Phi'''}$$

Certaines prémisses de ces trois règles utilisent l'une des notations :

$$\Phi, \{\ \Gamma \vdash \ldots\ \}_{i=1}^n \rhd \Phi' \qquad \text{ou} \qquad \Phi \vdash \{\ \ldots\ \}_{i=1}^n \rhd \Phi'$$

Elles désignent le typage successif de n expressions ou l'ajout de n contraintes de sous-typage selon le cas. L'ensemble de contraintes Φ est propagé à chaque étape. Une telle prémisse provoque la génération de n fils lors de l'instanciation de la règle correspondante dans un arbre d'inférence. Elles correspondent simplement à un « fold » dans l'implémentation du typeur.

La première règle (TMatch) sert de point d'entrée pour le typage d'un filtrage. Elle impose de typer l'expression à laquelle est appliquée le filtrage, et relie la variable de type α_e générée pour l'occasion avec le type `t` du **`GADT`** associé aux constructeurs mentionnés dans les cas de filtrage. Nous remarquerons que cette règle ne peut s'appliquer que si tous les constructeurs appartiennent au même type `t`. Nous faisons l'hypothèse que l'unicité des constructeurs a été vérifiée lors de la définition des types **`GADT`**, et qu'un mécanisme standard quelconque permet de retrouver le type **`GADT`** associé aux constructeurs mentionnés dans un filtrage.

Un filtrage de motif sur un **`GADT`** peut être « partiel » : tous les constructeurs du **`GADT`** ne sont pas obligatoirement mentionnés dans l'un des cas de filtrage. Pour simplifier l'écriture de la règle TMatch, nous avons supposé que les constructeurs ont été ordonnés pour que les premiers (dont l'indice varie de 1 à n avec $n \geqslant 1$) correspondent à un cas de filtrage, et les éventuels autres (dont l'indice varie de $n+1$ à s avec $s \geqslant n$) sont absents du filtrage. Le typage de chaque cas de filtrage est alors délégué à la règle TCase, et la gestion des constructeurs du **`GADT`** ne correspondant à aucun cas de filtrage est déléguée à la règle TAbsentCase.

Le typage d'un cas de filtrage, géré par la règle TCase, est plus subtil. Détaillons-en les différents points :

- Le « non-échappement des types existentiels » est géré grâce aux variables α^l et α^r. Celles-ci représentent les bornes des variables de type universelles générées lors du typage de l'expression e présente dans le cas de filtrage. Elles sont stockées dans les types existentiels générés et sont utilisées, comme nous l'avons expliqué précédemment et comme nous le verrons grâce aux règles de saturation à venir, lors de la comparaison entre un type existentiel et une variable universelle pour appliquer un traitement différent selon que la variable universelle a été générée lors du typage de e ou non.
- Les variables de type $\beta'_1, \ldots, \beta'_n$ sont des variables de type fraîches universelles classiques. Les types existentiels sont notés avec un ϵ.
- La notation $\overline{\Phi_1}$ désigne la négation de Φ_1 vu comme une formule logique. Si Φ_1 est une simple conjonction de relations de sous-typage et de relations de « non sous-typage », sa négation est bien une disjonction de relations de sous-typage et de relations de « non sous-typage », et est alors de la même forme que dans les systèmes précédents. Par contre, si Φ_1 contient des disjonctions, sa négation contient des conjonctions, d'où la définition de Ψ comme pouvant contenir des conjonctions.
- Pour chaque variable de type β_i, liée par un $\exists$ dans l'ensemble de contraintes associé au constructeur `K` lors de la définition du **`GADT`** correspondant, est créée une variable de type universelle β'_i qui est contrainte par une relation de double inégalité avec le type existentiel $\epsilon(\beta'_i, \Phi_1, [\alpha^l; \alpha^r], \{\beta'_1, \ldots, \beta'_q\})$ comme imposé par les deux dernières prémisses de la règle TCase. Les variables β_i étant générées entre α^l et α^r, le type existentiel ne « s'échappe pas de sa portée » à travers β'_i. Ces contraintes de sous-typage sont donc gérées comme dans les systèmes précédents. Les variables universelles β'_i ne sont donc que des « variables intermédiaires », et sont principalement utiles pour simplifier l'écriture des règles TCase et TAbsentCase et pour maintenir les types présents dans les contraintes sous forme normale : les paramètres des constructeurs de type sont toujours des variables de type universelles.

La règle TAbsentCase est très similaire à la règle TCase. Ceci est assez intuitif car il est équivalent de gérer l'absence d'un cas dans un filtrage et le typage d'un cas avec une expression qui ne type pas. La prémisse $\Phi, \Gamma \oplus \{\, \mathtt{x}_i : \beta'_i \,\}_{i=1}^{p} \vdash \Psi \vee \overline{\Phi_1} \vee \mathtt{e} : \alpha_e \rhd \Phi'$ de la règle TCase est alors simplement transformée en $\Phi \vdash \Psi \vee \overline{\Phi_1} \rhd \Phi'$ dans la règle TAbsentCase. En effet, le mécanisme de saturation générerait la contrainte $\Psi \vee \overline{\Phi_1}$ si le typage de $\mathtt{e}$ engendrait une contrainte invalide.

Bien évidemment, une implémentation concrète de ce système de types devrait générer un « warning » lorsque l'expression d'un cas de filtrage ne type pas. En effet, générer silencieusement une contrainte invalidant un tel cas peut être surprenant pour le programmeur, malgré le fait que ce comportement est parfaitement valide pour le système de types. Néanmoins, accepter les programmes contenant des filtrages comportant des cas avec une expression qui ne type pas est important d'un point de vue théorique pour que le système de types vérifie la propriété de « réduction du sujet ». Il est en effet parfaitement possible d'obtenir un programme comportant de tels filtrages après quelques β-réductions d'un programme n'en comportant pas.

5.3.4 Règle d'instanciation

La règle d'instanciation est identique à celle du chapitre 3 :

$$\textsc{Inst}\quad \frac{\texttt{let}\ \alpha'_1,\ldots,\alpha'_n\ \mathit{fresh}\quad \Phi \vdash \Psi \vee \alpha_0[\alpha_i \mapsto \alpha'_i]_{i=1}^n \leqslant \tau^r \rhd \Phi_1 \qquad \Phi_1 \vdash \Psi \vee \Psi_1[\alpha_i \mapsto \alpha'_i]_{i=1}^n \rhd \Phi_2 \quad \cdots \quad \Phi_p \vdash \Psi \vee \Psi_p[\alpha_i \mapsto \alpha'_i]_{i=1}^n \rhd \Phi_{p+1}}{\Phi \vdash \Psi \vee [\ \forall \alpha_1\ \ldots\ \alpha_n\ .\ \alpha_0 \mid \Psi_1 \wedge \cdots \wedge \Psi_p\ ] \leqslant \tau^r \rhd \Phi_{p+1}}$$

5.3.5 Règles de saturation

Gestion des conjonctions dans Ψ :

La présence de conjonctions dans les Ψ est simplement gérée par la règle de saturation suivante distribuant les contraintes liées par les conjonctions :

$$\textsc{SDistOrAnd}\quad \frac{\Phi_0 \vdash \Psi \vee \mathsf{C}_1 \rhd \Phi_1 \qquad \ldots \qquad \Phi_{n-1} \vdash \Psi \vee \mathsf{C}_n \rhd \Phi_n}{\Phi_0 \vdash \Psi \vee (\mathsf{C}_1 \wedge \cdots \wedge \mathsf{C}_n) \rhd \Phi_n}$$

Calcul de point fixe :

Le mécanisme de saturation des contraintes de sous-typage par recherche d'un point fixe est très similaire à celui des systèmes précédents :

$$\textsc{SNewConstraint}(\leqslant)\quad \frac{\texttt{when}\ (\Psi \vee \tau^l \leqslant \tau^r) \notin \Phi \qquad \Phi \wedge (\Psi \vee \tau^l \leqslant \tau^r) \vdash \Psi \vee \tau^l \leq \tau^r \rhd \Phi'}{\Phi \vdash \Psi \vee \tau^l \leqslant \tau^r \rhd \Phi'} \qquad \textsc{SAlreadyProved}(\leqslant)\quad \frac{\texttt{when}\ (\Psi \vee \tau^l \leqslant \tau^r) \in \Phi}{\Phi \vdash \Psi \vee \tau^l \leqslant \tau^r \rhd \Phi}$$

Les nouvelles relations de non-sous-typage sont gérées de manière similaire :

$$\textsc{SNewConstraint}(\not\leqslant)\quad \frac{\texttt{when}\ (\Psi \vee \tau^l \not\leqslant \tau^r) \notin \Phi \qquad \Phi \wedge (\Psi \vee \tau^l \not\leqslant \tau^r) \vdash \Psi \vee \tau^l \not\leq \tau^r \rhd \Phi'}{\Phi \vdash \Psi \vee \tau^l \not\leqslant \tau^r \rhd \Phi'} \qquad \textsc{SAlreadyProved}(\not\leqslant)\quad \frac{\texttt{when}\ (\Psi \vee \tau^l \not\leqslant \tau^r) \in \Phi}{\Phi \vdash \Psi \vee \tau^l \not\leqslant \tau^r \rhd \Phi}$$

Axiomes :

Il y a maintenant trois axiomes correspondant respectivement à une relation de sous-typage entre deux fois la même variable de type universelle, entre deux fois le même type existentiel, et une

relation de non-sous-typage entre deux types paramétrés de noms différents :

$$\textsc{SLeqSameVar}\ \frac{}{\Phi \vdash \Psi \vee \alpha \leqslant \alpha \rhd \Phi} \qquad \textsc{SNotLeqDiffParamed}\ \frac{\texttt{when t} \neq \texttt{u}}{\Phi \vdash \Psi \vee (\alpha_1, \ldots, \alpha_n)\ \texttt{t} \not\leq (\alpha'_1, \ldots, \alpha'_p)\ \texttt{u} \rhd \Phi}$$

$$\textsc{SLeqSameExist}\ \frac{}{\Phi \vdash \Psi \vee \epsilon(\alpha_0, \Phi_0, [\alpha^l; \alpha^r], \{\alpha_1, \ldots, \alpha_m\}) \leqslant \epsilon(\alpha_0, \Phi_0, [\alpha^l; \alpha^r], \{\alpha_1, \ldots, \alpha_m\}) \rhd \Phi}$$

Comparaisons entre types paramétrés :

Lorsqu'une relation de sous-typage relie deux types paramétrés de même nom, cette relation est reportée sur les paramètres dans les deux sens sous forme d'une conjonction. Pour une relation de non-sous-typage entre deux types paramétrés de même nom, la contrainte est également reportée sur les paramètres mais cette fois sous forme d'une disjonction. Le nombre de paramètres des deux types est par construction obligatoirement le même :

$$\textsc{SLeqSameParamed}\ \frac{\Phi \vdash \{\ \Psi \vee \alpha_i \leqslant \alpha'_i\ \}_{i=1}^n \rhd \Phi' \qquad \Phi' \vdash \{\ \Psi \vee \alpha'_i \leqslant \alpha_i\ \}_{i=1}^n \rhd \Phi''}{\Phi \vdash \Psi \vee (\alpha_1, \ldots, \alpha_n)\ \texttt{t} \leq (\alpha'_1, \ldots, \alpha'_n)\ \texttt{t} \rhd \Phi''}$$

$$\textsc{SNotLeqSameParamed}\ \frac{\Phi \vdash \Psi \ \vee\ \alpha_1 \not\leq \alpha'_1 \vee \cdots \vee \alpha_n \not\leq \alpha'_n \ \vee\ \alpha'_1 \not\leq \alpha_1 \vee \cdots \vee \alpha'_n \not\leq \alpha_n \rhd \Phi'}{\Phi \vdash \Psi \vee (\alpha_1, \ldots, \alpha_n)\ \texttt{t} \not\leq (\alpha'_1, \ldots, \alpha'_n)\ \texttt{t} \rhd \Phi'}$$

Règles de « backtrack » :

Les règles mettant en évidence les « clashs », duales des axiomes, reportent la saturation sur l'ensemble Ψ de contraintes alternatives lorsque celui-ci n'est pas vide :

$$\textsc{SNotLeqSameVar}\ \frac{\Phi \vdash \Psi \vee \phi \rhd \Phi'}{\Phi \vdash \Psi \vee \phi \vee \alpha \not\leq \alpha \rhd \Phi'} \qquad \textsc{SLeqDiffParamed}\ \frac{\Phi \vdash \Psi \vee \phi \rhd \Phi'}{\Phi \vdash \Psi \vee \phi \vee (\alpha_1, \ldots, \alpha_n)\ \texttt{t} \leq (\alpha'_1, \ldots, \alpha'_p)\ \texttt{u} \rhd \Phi'}$$

$$\textsc{SNotLeqSameExist}\ \frac{\Phi \vdash \Psi \vee \phi \rhd \Phi'}{\Phi \vdash \Psi \vee \phi \vee \epsilon(\alpha_0, \Phi_0, [\alpha^l; \alpha^r], \{\alpha_1, \ldots, \alpha_m\}) \not\leq \epsilon(\alpha_0, \Phi_0, [\alpha^l; \alpha^r], \{\alpha_1, \ldots, \alpha_m\}) \rhd \Phi'}$$

Gestion des relations de sous-typage faisant intervenir une variable de type universelle :

L'ajout d'une relation de sous-typage de la forme $\alpha \leqslant (\alpha_1, \ldots, \alpha_n)\ \mathtt{t}$ doit, en plus de prendre en compte la transitivité de $(\leqslant)$ comme précédemment, tenir compte de la présence d'éventuelles relations de non-sous-typage de la forme $\alpha \not\leqslant \tau^l$ dans l'ensemble Φ de contraintes déjà connues. Pour ce faire, nous utilisons les deux méta-fonctions LEFTS et $\overline{\text{RIGHTS}}$:

SVarLeqParamed

$$\frac{\begin{array}{cc}\mathtt{let}\ (\Psi_1, \tau_1^l), \ldots, (\Psi_p, \tau_p^l) = \textsc{lefts}(\alpha,\ \Phi) & \mathtt{let}\ (\Psi'_1, \tau_1'^l), \ldots, (\Psi'_q, \tau_q'^l) = \overline{\textsc{rights}}(\alpha,\ \Phi) \\ \Phi \vdash \{\ \Psi \vee \Psi_i \vee \tau_i^l \leqslant (\alpha_1, \ldots, \alpha_n)\ \mathtt{t}\ \}_{i=1}^{p} \rhd \Phi' & \Phi' \vdash \{\ \Psi \vee \Psi'_i \vee (\alpha_1, \ldots, \alpha_n)\ \mathtt{t} \not\leqslant \tau_i'^l\ \}_{i=1}^{q} \rhd \Phi''\end{array}}{\Phi \vdash \Psi \vee \alpha \leq (\alpha_1, \ldots, \alpha_n)\ \mathtt{t} \rhd \Phi''}$$

L'ajout d'une relation de sous-typage dans l'autre sens, c'est-à-dire de la forme $(\alpha_1, \ldots, \alpha_n)\ \mathtt{t} \leqslant \alpha$, est gérée de manière similaire par la règle suivante :

SParamedLeqVar

$$\frac{\begin{array}{cc}\mathtt{let}\ (\Psi_1, \tau_1^r), \ldots, (\Psi_p, \tau_p^r) = \textsc{rights}(\alpha,\ \Phi) & \mathtt{let}\ (\Psi'_1, \tau_1'^r), \ldots, (\Psi'_q, \tau_q'^r) = \overline{\textsc{lefts}}(\alpha,\ \Phi) \\ \Phi \vdash \{\ \Psi \vee \Psi_i \vee (\alpha_1, \ldots, \alpha_n)\ \mathtt{t} \leqslant \tau_i^r\ \}_{i=1}^{p} \rhd \Phi' & \Phi' \vdash \{\ \Psi \vee \Psi'_i \vee \tau_i'^r \not\leqslant (\alpha_1, \ldots, \alpha_n)\ \mathtt{t}\ \}_{i=1}^{q} \rhd \Phi''\end{array}}{\Phi \vdash \Psi \vee (\alpha_1, \ldots, \alpha_n)\ \mathtt{t} \leq \alpha \rhd \Phi''}$$

L'ajout d'une relation de sous-typage entre deux variables de type universelles, c'est-à-dire de la forme $\alpha_1 \leqslant \alpha_2$, se gère par une règle accumulant les prémisses des deux règles précédentes. Il faut alors, comme habituellement, extraire de Φ tous les types plus petits que α_1 et les lier par une relation de sous-typage avec tous les types plus grands que α_2. Par ailleurs, à cause de la présence de relations de non-sous-typage, il faut extraire tous les types plus grands que α_2 et les lier par une relation de non-sous-typage avec les types qui ne sont pas plus grands que α_1. Enfin, il faut extraire tous les types qui ne sont pas plus petits que α_2 et les lier par une relation de non-sous-typage avec les types qui sont plus petits que α_1. Ce comportement est représenté par la règle suivante :

SVarLeqVar

$$\frac{\begin{array}{c}\mathtt{when}\ \alpha_1 \neq \alpha_2 \\ \begin{array}{cc}\mathtt{let}\ (\Psi_1^l, \tau_1^l), \ldots, (\Psi_p^l, \tau_p^l) = \textsc{lefts}(\alpha_1,\ \Phi),\ (\varnothing, \alpha_1) & \mathtt{let}\ (\Psi_1'^l, \tau_1'^l), \ldots, (\Psi_q'^l, \tau_q'^l) = \overline{\textsc{rights}}(\alpha_1,\ \Phi) \\ \mathtt{let}\ (\Psi_1^r, \tau_1^r), \ldots, (\Psi_r^r, \tau_r^r) = \textsc{rights}(\alpha_2,\ \Phi),\ (\varnothing, \alpha_2) & \mathtt{let}\ (\Psi_1'^r, \tau_1'^r), \ldots, (\Psi_s'^r, \tau_s'^r) = \overline{\textsc{lefts}}(\alpha_2,\ \Phi)\end{array} \\ \Phi \vdash \{\ \{\ \Psi \vee \Psi_i^l \vee \Psi_j^r \vee \tau_i^l \leqslant \tau_j^r\ \}_{i=1}^{p}\ \}_{j=1}^{r} \rhd \Phi' \\ \Phi' \vdash \{\ \{\ \Psi \vee \Psi_i'^l \vee \Psi_j^r \vee \tau_j^r \not\leqslant \tau_i'^l\ \}_{i=1}^{q}\ \}_{j=1}^{r} \rhd \Phi'' \quad \Phi'' \vdash \{\ \{\ \Psi \vee \Psi_i^l \vee \Psi_j'^r \vee \tau_j'^r \not\leqslant \tau_i^l\ \}_{i=1}^{p}\ \}_{j=1}^{s} \rhd \Phi'''\end{array}}{\Phi \vdash \Psi \vee \alpha_1 \leq \alpha_2 \rhd \Phi'''}$$

L'introduction d'une relation de sous-typage entre deux fois la même variable est déjà gérée différemment par la règle SLeqSameVar, d'où la contrainte « $\mathtt{when}\ \alpha_1 \neq \alpha_2$ ».

La potentielle présence de types existentiels complexifie légèrement cette règle par l'ajout des $(\varnothing, \alpha_1)$ et $(\varnothing, \alpha_2)$ aux ensembles retournés par les méta-fonctions LEFTS et RIGHTS. Il n'est en effet

pas toujours valide de comparer une variable de type universelle avec un type existentiel et les variables universelles α_1 et α_2 doivent être prises en compte dans les comparaisons avec les différents types τ_i^l, $\tau_i'^l$, τ_i^r et $\tau_i'^r$.

Pour mettre en évidence ce phénomène, considérons l'exemple suivant : si Φ contient une contrainte de la forme $(\epsilon(\beta, \Phi_1, [\alpha^l; \alpha^r], \{\beta_1, \ldots, \beta_m\}) \leqslant \alpha_1)$, en cas d'ajout de la contrainte $(\alpha_1 \leqslant \alpha_2)$, lorsque $\alpha_2 \notin [\alpha^l; \alpha^r]$, il est nécessaire de générer la contrainte $(\epsilon(\beta, \Phi_1, [\alpha^l; \alpha^r], \{\beta_1, \ldots, \beta_m\}) \leqslant \alpha_2)$ pour gérer l'échappement de portée de $\epsilon(\beta, \Phi_1, [\alpha^l; \alpha^r], \{\beta_1, \ldots, \beta_m\})$.

La gestion d'une relation de sous-typage entre une variable de type universelle et un type existentiel est présentée plus loin.

Gestion des relations de non-sous-typage faisant intervenir une variable de type universelle :

L'ajout de contraintes de non-sous-typage entre une variable de type et un type paramétré est en fait plus simple que l'ajout d'une contrainte de sous-typage car il n'a pas à tenir compte de la présence d'autres relations de non-sous-typage dans Φ, seulement de la présence de relations de sous-typage faisant intervenir la même variable de type. Il est géré par les deux règles suivantes :

SVarNotLeqParamed

$$\frac{\texttt{let}\ (\Psi_1, \tau_1^r), \ldots, (\Psi_p, \tau_p^r) = \textsc{rights}(\alpha,\ \Phi) \qquad \Phi \vdash \{\ \Psi \vee \Psi_i \vee \tau_i^r \not\leqslant (\alpha_1, \ldots, \alpha_n)\ \texttt{t}\ \}_{i=1}^{p} \rhd \Phi'}{\Phi \vdash \Psi \vee \alpha \not\leqslant (\alpha_1, \ldots, \alpha_n)\ \texttt{t} \rhd \Phi'}$$

SParamedNotLeqVar

$$\frac{\texttt{let}\ (\Psi_1, \tau_1^l), \ldots, (\Psi_p, \tau_p^l) = \textsc{lefts}(\alpha,\ \Phi) \qquad \Phi \vdash \{\ \Psi \vee \Psi_i \vee (\alpha_1, \ldots, \alpha_n)\ \texttt{t} \not\leqslant \tau_i^l\ \}_{i=1}^{p} \rhd \Phi'}{\Phi \vdash \Psi \vee (\alpha_1, \ldots, \alpha_n)\ \texttt{t} \not\leqslant \alpha \rhd \Phi'}$$

De la même manière que la règle SVarLeqVar était obtenue en mixant les règles SVarLeqParamed et SParamedLeqVar, la règle gérant l'ajout d'une contrainte de non-sous-typage entre deux variables de type universelles est obtenue en mixant les deux règles précédentes :

SVarNotLeqVar

$$\frac{\begin{array}{c}\texttt{when}\ \alpha_1 \neq \alpha_2 \\ \texttt{let}\ (\Psi_1^r, \tau_1^r), \ldots, (\Psi_p^r, \tau_p^r) = \textsc{rights}(\alpha_1,\ \Phi),\ (\varnothing, \alpha_1) \\ \texttt{let}\ (\Psi_1^l, \tau_1^l), \ldots, (\Psi_q^l, \tau_q^l) = \textsc{lefts}(\alpha_2,\ \Phi),\ (\varnothing, \alpha_2) \\ \Phi \vdash \{\ \{\ \Psi \vee \Psi_i^r \vee \Psi_j^l \vee \tau_i^r \not\leqslant \tau_j^l\ \}_{i=1}^{p}\ \}_{j=1}^{q} \rhd \Phi'\end{array}}{\Phi \vdash \Psi \vee \alpha_1 \not\leqslant \alpha_2 \rhd \Phi'}$$

L'introduction d'une relation de non-sous-typage entre deux fois la même variable est déjà gérée différemment par la règle SNotLeqSameVar, d'où la contrainte « $\texttt{when}\ \alpha_1 \neq \alpha_2$ ».

Tout comme pour la règle SVarLeqVar, les couples $(\varnothing, \alpha_1)$ et $(\varnothing, \alpha_2)$ doivent être ajoutés aux retours des méta-fonctions lefts et rights pour prendre en compte la potentielle présence de types existentiels.

La gestion d'une relation de non-sous-typage entre une variable de type universelle et un type existentiel est présentée plus loin.

Comparaison entre un type existentiel et une variable universelle dans sa portée :

Une relation de sous-typage faisant intervenir au moins un type existentiel doit être gérée de manière particulière, sauf lorsque cette comparaison se fait avec une variable de type universelle qui est dans la portée du type existentiel. Dans ce cas, le type existentiel est géré exactement comme un type paramétré. Une telle situation est gérée par les deux règles suivantes, très similaires aux règles SVarLeqParamed et SParamedLeqVar :

SVarLeqExistIn

$$\frac{\begin{array}{c}\texttt{when } \alpha \in [\alpha^l;\alpha^r] \\ \texttt{let } (\Psi_1,\tau_1^l),\ldots,(\Psi_p,\tau_p^l) = \textsc{lefts}(\alpha,\ \Phi) \qquad \texttt{let } (\Psi'_1,\tau_1'^l),\ldots,(\Psi'_q,\tau_q'^l) = \overline{\textsc{rights}}(\alpha,\ \Phi) \\ \Phi \vdash \{\ \Psi \vee \Psi_i \vee \tau_i^l \leqslant \epsilon(\alpha_0,\Phi_0,[\alpha^l;\alpha^r],\{\alpha_1,\ldots,\alpha_m\})\ \}_{i=1}^{p} \rhd \Phi' \\ \Phi' \vdash \{\ \Psi \vee \Psi'_i \vee \epsilon(\alpha_0,\Phi_0,[\alpha^l;\alpha^r],\{\alpha_1,\ldots,\alpha_m\}) \nleqslant \tau_i'^l\ \}_{i=1}^{q} \rhd \Phi''\end{array}}{\Phi \vdash \Psi \vee \alpha \leq \epsilon(\alpha_0,\Phi_0,[\alpha^l;\alpha^r],\{\alpha_1,\ldots,\alpha_m\}) \rhd \Phi''}$$

SExistLeqVarIn

$$\frac{\begin{array}{c}\texttt{when } \alpha \in [\alpha^l;\alpha^r] \\ \texttt{let } (\Psi_1,\tau_1^r),\ldots,(\Psi_p,\tau_p^r) = \textsc{rights}(\alpha,\ \Phi) \qquad \texttt{let } (\Psi'_1,\tau_1'^r),\ldots,(\Psi'_q,\tau_q'^r) = \overline{\textsc{lefts}}(\alpha,\ \Phi) \\ \Phi \vdash \{\ \Psi \vee \Psi_i \vee \epsilon(\alpha_0,\Phi_0,[\alpha^l;\alpha^r],\{\alpha_1,\ldots,\alpha_m\}) \leqslant \tau_i^r\ \}_{i=1}^{p} \rhd \Phi' \\ \Phi' \vdash \{\ \Psi \vee \Psi'_i \vee \tau_i'^r \nleqslant \epsilon(\alpha_0,\Phi_0,[\alpha^l;\alpha^r],\{\alpha_1,\ldots,\alpha_m\})\ \}_{i=1}^{q} \rhd \Phi''\end{array}}{\Phi \vdash \Psi \vee \epsilon(\alpha_0,\Phi_0,[\alpha^l;\alpha^r],\{\alpha_1,\ldots,\alpha_m\}) \leq \alpha \rhd \Phi''}$$

De même, lorsque la relation en question est une relation de non-sous-typage, elle est gérée par les deux règles suivantes, similaires à SVarNotLeqParamed et SParamedNotLeqVar :

SVarNotLeqExistIn

$$\frac{\begin{array}{c}\texttt{when } \alpha \in [\alpha^l;\alpha^r] \qquad \texttt{let } (\Psi_1,\tau_1^r),\ldots,(\Psi_p,\tau_p^r) = \textsc{rights}(\alpha,\ \Phi) \\ \Phi \vdash \{\ \Psi \vee \Psi_i \vee \tau_i^r \nleqslant \epsilon(\alpha_0,\Phi_0,[\alpha^l;\alpha^r],\{\alpha_1,\ldots,\alpha_m\})\ \}_{i=1}^{p} \rhd \Phi'\end{array}}{\Phi \vdash \Psi \vee \alpha \nleq \epsilon(\alpha_0,\Phi_0,[\alpha^l;\alpha^r],\{\alpha_1,\ldots,\alpha_m\}) \rhd \Phi'}$$

SExistNotLeqVarIn

$$\frac{\begin{array}{c}\texttt{when } \alpha \in [\alpha^l;\alpha^r] \qquad \texttt{let } (\Psi_1,\tau_1^l),\ldots,(\Psi_p,\tau_p^l) = \textsc{lefts}(\alpha,\ \Phi) \\ \Phi \vdash \{\ \Psi \vee \Psi_i \vee \epsilon(\alpha_0,\Phi_0,[\alpha^l;\alpha^r],\{\alpha_1,\ldots,\alpha_m\}) \nleqslant \tau_i^l\ \}_{i=1}^{p} \rhd \Phi'\end{array}}{\Phi \vdash \Psi \vee \epsilon(\alpha_0,\Phi_0,[\alpha^l;\alpha^r],\{\alpha_1,\ldots,\alpha_m\}) \nleq \alpha \rhd \Phi'}$$

Comparaison entre un type existentiel et une variable universelle hors de sa portée :

En revanche, lorsque la comparaison fait intervenir un type existentiel et une variable de type universelle en dehors de la portée du type existentiel, comme nous l'avons décrit dans la section

précédente, la comparaison est reportée sur les éventuels autres types extraits des contraintes provenant de la déclaration du constructeur de données ayant engendré le type existentiel en question. Lorsque cette relation est une relation de sous-typage, ce comportement est implémenté par les deux règles suivantes :

$$\textsc{SVarLeqExistOut}\quad \frac{\texttt{when}\ \alpha \notin [\alpha^l;\alpha^r] \qquad \texttt{let}\ (\Psi_1,\tau_1^l),\dots,(\Psi_n,\tau_n^l) = \not\epsilon\textsc{lefts}(\alpha_0,\Phi_0,\{\alpha_1,\dots,\alpha_m\}) \qquad \Phi \vdash \Psi \vee (\overline{\Psi_1} \wedge \alpha \leqslant \tau_1^l) \vee \dots \vee (\overline{\Psi_n} \wedge \alpha \leqslant \tau_n^l) \rhd \Phi'}{\Phi \vdash \Psi \vee \alpha \leq \epsilon(\alpha_0,\Phi_0,[\alpha^l;\alpha^r],\{\alpha_1,\dots,\alpha_m\}) \rhd \Phi'}$$

$$\textsc{SExistLeqVarOut}\quad \frac{\texttt{when}\ \alpha \notin [\alpha^l;\alpha^r] \qquad \texttt{let}\ (\Psi_1,\tau_1^r),\dots,(\Psi_n,\tau_n^r) = \not\epsilon\textsc{rights}(\alpha_0,\Phi_0,\{\alpha_1,\dots,\alpha_m\}) \qquad \Phi \vdash \Psi \vee (\overline{\Psi_1} \wedge \tau_1^r \leqslant \alpha) \vee \dots \vee (\overline{\Psi_n} \wedge \tau_n^r \leqslant \alpha) \rhd \Phi'}{\Phi \vdash \Psi \vee \epsilon(\alpha_0,\Phi_0,[\alpha^l;\alpha^r],\{\alpha_1,\dots,\alpha_m\}) \leq \alpha \rhd \Phi'}$$

Dans la règle SVarLeqExist, les Ψ_i, lorsqu'ils ne sont pas vides, représentent, dans l'ensemble de contraintes original défini par le programmeur lors de la définition du **GADT** correspondant, des contraintes dont l'une peut être vraie à la place de $\tau_i^l \leqslant \alpha_0$. Pour assurer que $\tau_i^l \leqslant \alpha_0$ est vrai, il faut donc imposer la négation de Ψ_i, d'où les $(\overline{\Psi_i} \wedge \dots)$. Idem pour la règle SExistLeqVar.

L'usage des fonctions $\not\epsilon$lefts et $\not\epsilon$rights permet d'ignorer de Φ_0 les contraintes reliant α_0 à une variable correspondant à un autre type existentiel. Ceci est primordial pour la validité du système.

Les relations de non-sous-typage faisant intervenir un type existentiel sont plus compliquées à gérer car il existe deux façons de déduire une relation de non-sous-typage : soit grâce à une relation de sous-typage venant de Φ_0 et en imposant une relation de non-sous-typage, soit grâce à une relation de non-sous-typage provenant de Φ_0 et en imposant une relation de sous-typage. Ceci est formalisé par les deux règles suivantes :

$$\textsc{SVarNotLeqExistOut}\quad \frac{\begin{array}{c}\texttt{when}\ \alpha \notin [\alpha^l;\alpha^r] \qquad \texttt{let}\ (\Psi_1,\tau_1^r),\dots,(\Psi_n,\tau_n^r) = \not\epsilon\textsc{rights}(\alpha_0,\Phi_0,\{\alpha_1,\dots,\alpha_m\}) \\ \texttt{let}\ (\Psi'_1,\tau'^r_1),\dots,(\Psi'_p,\tau'^r_p) = \overline{\not\epsilon\textsc{lefts}}(\alpha_0,\Phi_0,\{\alpha_1,\dots,\alpha_m\}) \\ \Phi \vdash \Psi \ \vee\ (\overline{\Psi_1} \wedge \alpha \not\leqslant \tau_1^r) \vee \dots \vee (\overline{\Psi_n} \wedge \alpha \not\leqslant \tau_n^r) \ \vee\ (\overline{\Psi'_1} \wedge \tau'^r_1 \leqslant \alpha) \vee \dots \vee (\overline{\Psi'_p} \wedge \tau'^r_p \leqslant \alpha) \rhd \Phi'\end{array}}{\Phi \vdash \Psi \vee \alpha \not\leq \epsilon(\alpha_0,\Phi_0,[\alpha^l;\alpha^r],\{\alpha_1,\dots,\alpha_m\}) \rhd \Phi'}$$

$$\textsc{SExistNotLeqVarOut}\quad \frac{\begin{array}{c}\texttt{when}\ \alpha \notin [\alpha^l;\alpha^r] \qquad \texttt{let}\ (\Psi_1,\tau_1^l),\dots,(\Psi_n,\tau_n^l) = \not\epsilon\textsc{lefts}(\alpha_0,\Phi_0,\{\alpha_1,\dots,\alpha_m\}) \\ \texttt{let}\ (\Psi'_1,\tau'^l_1),\dots,(\Psi'_p,\tau'^l_p) = \overline{\not\epsilon\textsc{rights}}(\alpha_0,\Phi_0,\{\alpha_1,\dots,\alpha_m\}) \\ \Phi \vdash \Psi \ \vee\ (\overline{\Psi_1} \wedge \tau_1^l \not\leqslant \alpha) \vee \dots \vee (\overline{\Psi_n} \wedge \tau_n^l \not\leqslant \alpha) \ \vee\ (\overline{\Psi'_1} \wedge \alpha \leqslant \tau'^l_1) \vee \dots \vee (\overline{\Psi'_p} \wedge \alpha \leqslant \tau'^l_p) \rhd \Phi'\end{array}}{\Phi \vdash \Psi \vee \epsilon(\alpha_0,\Phi_0,[\alpha^l;\alpha^r],\{\alpha_1,\dots,\alpha_m\}) \not\leq \alpha \rhd \Phi'}$$

Comparaison entre un type existentiel et un type paramétré :

Une relation de sous-typage ou de non-sous-typage entre un type existentiel et un type paramétré est, comme nous l'avons dit précédemment, géré exactement de la même manière que la comparaison entre un type existentiel et une variable de type universelle en dehors de sa portée.

Tout comme dans les règles précédentes, il est primordial pour la validité du système d'ignorer de Φ_0 les contraintes reliant α_0 à une variable de type correspondant à un autre type existentiel. Ce comportement est assuré par l'usage des fonctions $\notin$LEFTS, $\notin$RIGHTS, $\overline{\notin\text{LEFTS}}$ et $\overline{\notin\text{RIGHTS}}$.

Les quatre règles suivantes sont donc identiques aux quatre précédentes, en dehors du fait qu'elles font intervenir un type paramétré à la place d'une variable de type en dehors de la portée du type existentiel :

SPARAMEDLEQEXIST

$$\frac{\begin{array}{c}\texttt{let}\ (\Psi_1, \tau_1^l), \ldots, (\Psi_n, \tau_n^l) = \notin\text{LEFTS}(\alpha_0, \Phi_0, \{\alpha_1, \ldots, \alpha_m\}) \\ \Phi \vdash \Psi \vee (\overline{\Psi_1} \wedge (\alpha'_1, \ldots, \alpha'_q)\ \texttt{t} \leqslant \tau_1^l) \vee \cdots \vee (\overline{\Psi_n} \wedge (\alpha'_1, \ldots, \alpha'_q)\ \texttt{t} \leqslant \tau_n^l) \rhd \Phi'\end{array}}{\Phi \vdash \Psi \vee (\alpha'_1, \ldots, \alpha'_q)\ \texttt{t} \leq \epsilon(\alpha_0, \Phi_0, [\alpha^l; \alpha^r], \{\alpha_1, \ldots, \alpha_m\}) \rhd \Phi'}$$

SEXISTLEQPARAMED

$$\frac{\begin{array}{c}\texttt{let}\ (\Psi_1, \tau_1^r), \ldots, (\Psi_n, \tau_n^r) = \notin\text{RIGHTS}(\alpha_0, \Phi_0, \{\alpha_1, \ldots, \alpha_m\}) \\ \Phi \vdash \Psi \vee (\overline{\Psi_1} \wedge \tau_1^r \leqslant (\alpha'_1, \ldots, \alpha'_q)\ \texttt{t}) \vee \cdots \vee (\overline{\Psi_n} \wedge \tau_n^r \leqslant (\alpha'_1, \ldots, \alpha'_q)\ \texttt{t}) \rhd \Phi'\end{array}}{\Phi \vdash \Psi \vee \epsilon(\alpha_0, \Phi_0, [\alpha^l; \alpha^r], \{\alpha_1, \ldots, \alpha_m\}) \leq (\alpha'_1, \ldots, \alpha'_q)\ \texttt{t} \rhd \Phi'}$$

SPARAMEDNOTLEQEXIST

$$\frac{\begin{array}{c}\texttt{let}\ (\Psi_1, \tau_1^r), \ldots, (\Psi_n, \tau_n^r) = \notin\text{RIGHTS}(\alpha_0, \Phi_0, \{\alpha_1, \ldots, \alpha_m\}) \\ \texttt{let}\ (\Psi'_1, \tau_1'^r), \ldots, (\Psi'_p, \tau_p'^r) = \overline{\notin\text{LEFTS}}(\alpha_0, \Phi_0, \{\alpha_1, \ldots, \alpha_m\}) \\ \Phi \vdash \Psi \ \vee\ (\overline{\Psi_1} \wedge (\alpha'_1, \ldots, \alpha'_q)\ \texttt{t} \not\leqslant \tau_1^r) \vee \cdots \vee (\overline{\Psi_n} \wedge (\alpha'_1, \ldots, \alpha'_q)\ \texttt{t} \not\leqslant \tau_n^r) \vee \\ (\overline{\Psi'_1} \wedge \tau_1'^r \leqslant (\alpha'_1, \ldots, \alpha'_q)\ \texttt{t}) \vee \cdots \vee (\overline{\Psi'_p} \wedge \tau_p'^r \leqslant (\alpha'_1, \ldots, \alpha'_q)\ \texttt{t}) \rhd \Phi'\end{array}}{\Phi \vdash \Psi \vee (\alpha'_1, \ldots, \alpha'_q)\ \texttt{t} \not\leq \epsilon(\alpha_0, \Phi_0, [\alpha^l; \alpha^r], \{\alpha_1, \ldots, \alpha_m\}) \rhd \Phi'}$$

SEXISTNOTLEQPARAMED

$$\frac{\begin{array}{c}\texttt{let}\ (\Psi_1, \tau_1^l), \ldots, (\Psi_n, \tau_n^l) = \notin\text{LEFTS}(\alpha_0, \Phi_0, \{\alpha_1, \ldots, \alpha_m\}) \\ \texttt{let}\ (\Psi'_1, \tau_1'^l), \ldots, (\Psi'_p, \tau_p'^l) = \overline{\notin\text{RIGHTS}}(\alpha_0, \Phi_0, \{\alpha_1, \ldots, \alpha_m\}) \\ \Phi \vdash \Psi \ \vee\ (\overline{\Psi_1} \wedge \tau_1^l \not\leqslant (\alpha'_1, \ldots, \alpha'_q)\ \texttt{t}) \vee \cdots \vee (\overline{\Psi_n} \wedge \tau_n^l \not\leqslant (\alpha'_1, \ldots, \alpha'_q)\ \texttt{t}) \vee \\ (\overline{\Psi'_1} \wedge (\alpha'_1, \ldots, \alpha'_q)\ \texttt{t} \leqslant \tau_1'^l) \vee \cdots \vee (\overline{\Psi'_p} \wedge (\alpha'_1, \ldots, \alpha'_q)\ \texttt{t} \leqslant \tau_p'^l) \rhd \Phi'\end{array}}{\Phi \vdash \Psi \vee \epsilon(\alpha_0, \Phi_0, [\alpha^l; \alpha^r], \{\alpha_1, \ldots, \alpha_m\}) \not\leq (\alpha'_1, \ldots, \alpha'_q)\ \texttt{t} \rhd \Phi'}$$

Comparaison entre deux types existentiels :

L'introduction d'une relation de sous-typage et de non-sous-typage entre deux fois le même type existentiel est déjà gérée par les règles SLEQSAMEEXIST et SNOTLEQSAMEEXIST.

Lorsque les types existentiels sont différents, la contrainte est reportée sur les types liés par la même relation dans les ensembles de contraintes définis par le programmeur lors de la définition

des constructeurs de données correspondants. Une relation de sous-typage est alors gérée par la règle suivante :

SExistLeqExist

$$\frac{\begin{array}{c}\texttt{when } \alpha_0 \neq \alpha'_0 \\ \texttt{let } (\Psi_1, \tau_1^r), \ldots, (\Psi_n, \tau_n^r) = \textsc{rights}(\alpha_0, \Phi_0) \qquad \texttt{let } (\Psi'_1, \tau_1'^l), \ldots, (\Psi'_p, \tau_p'^l) = \textsc{lefts}(\alpha'_0, \Phi'_0) \\ \Phi \vdash \Psi \ \vee\ (\overline{\Psi_1} \wedge \overline{\Psi'_1} \wedge \tau_1^r \leqslant \tau_1'^l) \vee \cdots \vee (\overline{\Psi_1} \wedge \overline{\Psi'_p} \wedge \tau_1^r \leqslant \tau_p'^l) \\ \vdots \qquad\qquad\qquad\qquad \vdots \\ \vee\ (\overline{\Psi_n} \wedge \overline{\Psi'_1} \wedge \tau_n^r \leqslant \tau_1'^l) \vee \cdots \vee (\overline{\Psi_n} \wedge \overline{\Psi'_p} \wedge \tau_n^r \leqslant \tau_p'^l) \rhd \Phi'\end{array}}{\Phi \vdash \Psi \vee \epsilon(\alpha_0, \Phi_0, [\alpha^l; \alpha^r], \{\alpha_1, \ldots, \alpha_m\}) \leq \epsilon(\alpha'_0, \Phi'_0, [\alpha'_l; \alpha'_r], \{\alpha'_1, \ldots, \alpha'_{m'}\}) \rhd \Phi'}$$

et une relation de non-sous-typage par la règle suivante :

SExistNotLeqExist

$$\frac{\begin{array}{c}\texttt{when } \alpha_0 \neq \alpha'_0 \\ \texttt{let } (\Psi_1^l, \tau_1^l), \ldots, (\Psi_n^l, \tau_n^l) = \textsc{lefts}(\alpha_0, \Phi_0) \qquad \texttt{let } (\Psi_1'^l, \tau_1'^l), \ldots, (\Psi_p'^l, \tau_p'^l) = \overline{\textsc{rights}}(\alpha'_0, \Phi'_0) \\ \texttt{let } (\Psi_1^r, \tau_1^r), \ldots, (\Psi_q^r, \tau_q^r) = \textsc{rights}(\alpha'_0, \Phi'_0) \qquad \texttt{let } (\Psi_1'^r, \tau_1'^r), \ldots, (\Psi_s'^r, \tau_s'^r) = \overline{\textsc{lefts}}(\alpha'_0, \Phi'_0) \\ \Phi \vdash \Psi \ \vee\ (\overline{\Psi_1^l} \wedge \overline{\Psi_1^r} \wedge \tau_1^l \leqslant \tau_1^r) \vee \cdots \vee (\overline{\Psi_1^l} \wedge \overline{\Psi_q^r} \wedge \tau_1^l \leqslant \tau_q^r) \\ \vdots \qquad\qquad\qquad\qquad \vdots \\ \vee\ (\overline{\Psi_n^l} \wedge \overline{\Psi_1^r} \wedge \tau_n^l \leqslant \tau_1^r) \vee \cdots \vee (\overline{\Psi_n^l} \wedge \overline{\Psi_q^r} \wedge \tau_n^l \leqslant \tau_q^r) \\ \vee\ (\overline{\Psi_1'^l} \wedge \overline{\Psi_1^r} \wedge \tau_1'^l \not\leqslant \tau_1^r) \vee \cdots \vee (\overline{\Psi_1'^l} \wedge \overline{\Psi_q^r} \wedge \tau_1'^l \not\leqslant \tau_q^r) \\ \vdots \qquad\qquad\qquad\qquad \vdots \\ \vee\ (\overline{\Psi_p'^l} \wedge \overline{\Psi_1^r} \wedge \tau_p'^l \not\leqslant \tau_1^r) \vee \cdots \vee (\overline{\Psi_p'^l} \wedge \overline{\Psi_q^r} \wedge \tau_p'^l \not\leqslant \tau_q^r) \\ \vee\ (\overline{\Psi_1^l} \wedge \overline{\Psi_1'^r} \wedge \tau_1^l \not\leqslant \tau_1'^r) \vee \cdots \vee (\overline{\Psi_1^l} \wedge \overline{\Psi_s'^r} \wedge \tau_1^l \not\leqslant \tau_s'^r) \\ \vdots \qquad\qquad\qquad\qquad \vdots \\ \vee\ (\overline{\Psi_n^l} \wedge \overline{\Psi_1'^r} \wedge \tau_n^l \not\leqslant \tau_1'^r) \vee \cdots \vee (\overline{\Psi_n^l} \wedge \overline{\Psi_s'^r} \wedge \tau_n^l \not\leqslant \tau_s'^r) \rhd \Phi'\end{array}}{\Phi \vdash \Psi \vee \epsilon(\alpha_0, \Phi_0, [\alpha^l; \alpha^r], \{\alpha_1, \ldots, \alpha_m\}) \not\leq \epsilon(\alpha'_0, \Phi'_0, [\alpha'_l; \alpha'_r], \{\alpha'_1, \ldots, \alpha'_{m'}\}) \rhd \Phi'}$$

Nous remarquerons que, dans le cas particulier où les deux types existentiels $\epsilon(\alpha_0, \Phi_0, [\alpha^l; \alpha^r], \{\alpha_1, \ldots, \alpha_m\})$ et $\epsilon(\alpha'_0, \Phi'_0, [\alpha'_l; \alpha'_r], \{\alpha'_1, \ldots, \alpha'_{m'}\})$ proviennent du typage du même cas de filtrage (nous avons donc $\Phi_0 = \Phi'_0$, $\alpha^l = \alpha'_l$, $\alpha^r = \alpha'_r$ et $\{\alpha_1, \ldots, \alpha_m\} = \{\alpha'_1, \ldots, \alpha'_{m'}\}$) et lorsque les variables α_0 et α'_0 correspondantes sont reliées dans l'ensemble de contraintes Φ_0 par la même relation que celle (reliant les types existentiels) que l'on souhaite montrer valide et compatible avec les autres contraintes de Φ, la disjonction générée (par application de l'une des deux règles précédentes) contient obligatoirement les axiomes $(\epsilon(\alpha_0, \Phi_0, [\alpha^l; \alpha^r], \{\alpha_1, \ldots, \alpha_m\}) \leqslant \epsilon(\alpha_0, \Phi_0, [\alpha^l; \alpha^r], \{\alpha_1, \ldots, \alpha_m\}))$ et $(\epsilon(\alpha'_0, \Phi'_0, [\alpha'_l; \alpha'_r], \{\alpha'_1, \ldots, \alpha'_{m'}\}) \leqslant \epsilon(\alpha'_0, \Phi'_0, [\alpha'_l; \alpha'_r], \{\alpha'_1, \ldots, \alpha'_{m'}\}))$.

Plus généralement, il est facile de vérifier que lors de l'ajout d'une contrainte faisant intervenir un type existentiel $\epsilon(\alpha_0, \Phi_0, [\alpha^l; \alpha^r], \{\alpha_1, \ldots, \alpha_m\})$, lorsque cette contrainte (dans laquelle le type existentiel est remplacé par α_0) est membre de la conjonction Φ_0, toutes les disjonctions engendrées par saturation contiennent au moins une relation triviale de la forme $(\tau \leqslant \tau)$. Par exemple, l'ajout

d'une contrainte de la forme :

$$\epsilon(\alpha_0, (\cdots \wedge \alpha_0 \not\leqslant \texttt{int} \wedge \ldots), [\alpha^l; \alpha^r], \{\alpha_1, \ldots, \alpha_m\}) \not\leqslant \texttt{int}$$

engendrera des disjonctions contenant toutes ($\texttt{int} \leqslant \texttt{int}$). Un tel ajout ne peut ainsi jamais provoquer de clash.

Bien entendu, l'ajout d'une contrainte non-présente dans Φ_0 n'engendre pas toujours de clash. Nous en avons vu des exemples dans la section précédente. Il arrive que la contrainte imposée sur un type existentiel soit simplement reportée sur d'autres types, par exemple via un paramètre du `GADT`. Les règles que nous avons définies ici pour gérer l'ajout d'une contrainte faisant intervenir un type existentiel sont donc bien strictement plus puissantes qu'un simple test d'appartenance de cette contrainte à la conjonction de contraintes mentionnée par le programmeur lors de la définition du constructeur de données correspondant.

5.4 Adaptation de la preuve de terminaison

Contrairement au système du chapitre précédent dans lequel des variables de type pouvaient être générées lors de la saturation des contraintes, ce qui complexifiait la terminaison de l'algorithme d'inférence, le système présenté dans ce chapitre est beaucoup plus « classique ». La preuve de terminaison ne pose en conséquence aucun souci. Elle est en effet très similaire à celle du chapitre 3.

La partie typage de l'arbre d'inférence est isomorphe à la structure de l'expression que l'on souhaite typer, et est donc de taille finie. Le nombre de noeuds d'instanciation est au plus égal au nombre de noeuds de typage car ce sont les seuls qui peuvent avoir une prémisse de la forme d'une conclusion d'une règle d'instanciation. Les noeuds d'instanciation sont donc également en nombre fini.

Les types existentiels ne sont générés que par des noeuds de typage, elles sont donc également en nombre fini. La structure des autres types n'est pas récursive (car leur grammaire n'est pas récursive), et comme aucune règle de saturation ne génère de variables de type universelles, les types présents dans les sous-arbres de saturation sont en nombre fini. Le nombre de contraintes de sous-typage et de non-sous-typage est donc également fini, ainsi que le nombre de conjonctions, de disjonctions, de conjonctions de disjonctions et de disjonctions de conjonctions de contraintes. La taille de l'ensemble de contraintes Φ que l'on cherche à saturer par point fixe est donc bornée.

En conclusion, les sous-arbres de saturation sont en nombre fini et chacun de taille finie. L'algorithme d'inférence termine.

5.5 Adaptation de la preuve de validité

Nous partons de la preuve de validité du chapitre 3. Le système qui y est défini présente en effet de multiples similarités avec celui de ce chapitre de par la forme des règles d'inférence, et en

particulier à cause de la propagation d'un ensemble de contraintes alternatives Ψ à travers elles. Nous ne donnons donc ici que les modifications à apporter à la section 3.6 pour obtenir une preuve complète de validité de notre nouveau système.

5.5.1 Mise à jour des définitions de base

Nous commençons par redéfinir les notions de « validité » et de « saturation » d'un ensemble de contraintes vis-à-vis du nouveau formalisme introduit dans ce chapitre. Les ensembles de contraintes à la fois valides et saturés jouent en effet un rôle important dans cette preuve. Comme habituellement, ces deux définitions peuvent se déduire directement des règles d'inférence. Il s'agit en réalité d'une manière « non-constructive » de décrire les ensembles de contraintes que l'on souhaite obtenir par saturation, la version « constructive » étant donnée par les règles d'inférence.

La définition de la « validité » d'un ensemble de contraintes doit maintenant prendre en compte la présence de la relation de non-sous-typage ($\not\leqslant$). Nous définissons donc la notion de validité d'une relation de non-sous-typage entre deux types de manière très similaire à la validité d'une relation de sous-typage.

Définition 13 (Validité d'une contrainte « $\tau^l \not\leqslant \tau^r$ »)**.**

Une contrainte de type $\tau^l \not\leqslant \tau^r$ est dite « valide » s'il existe une règle de saturation dont la conclusion est de la forme $\Phi \vdash \tau^l \not\leqslant \tau^r \rhd \Phi'$, autrement dit s'il est possible d'effectuer au moins un pas de saturation à partir de $\Phi \vdash \tau^l \not\leqslant \tau^r \rhd \Phi'$.

La définition de la « saturation » d'un ensemble de contraintes doit également être modifiée pour prendre en compte la présence de conjonctions dans Ψ, la nouvelle relation ($\not\leqslant$) et la nouvelle construction de type ($\epsilon(\alpha, \Phi, [\alpha^l; \alpha^r], \{\alpha_1, \ldots, \alpha_m\})$).

Définition 3 (Saturation d'un ensemble « Φ » de contraintes)**.**

Un ensemble de contraintes Φ est dit « saturé » s'il vérifie toutes les propriétés suivantes :

- *Distribution de la conjonction sur la disjonction*
  (cf. SDistOrAnd*) :*
  - $\forall \Psi, C_1, \ldots, C_n \ .$
    $(\Psi \vee (C_1 \wedge \cdots \wedge C_n)) \in \Phi \ \Rightarrow$
    $(\Psi \vee C_1) \in \Phi \wedge \ldots \wedge (\Psi \vee C_n) \in \Phi$

- *Comparaison entre types paramétrés de même nom*
  (cf. SLeqSameParamed, SNotLeqSameParamed*) :*
  - $\forall \Psi, \alpha_1, \ldots, \alpha_n, \mathtt{t}, \alpha'_1, \ldots, \alpha'_n \ .$
    $(\Psi \vee (\alpha_1, \ldots, \alpha_n)\ \mathtt{t} \leqslant (\alpha'_1, \ldots, \alpha'_n)\ \mathtt{t}) \in \Phi \ \Rightarrow$

$\wedge\ (\Psi \vee \alpha_1 \leqslant \alpha'_1) \in \Phi \wedge \ldots \wedge (\Psi \vee \alpha_n \leqslant \alpha'_n) \in \Phi \wedge$
$\wedge\ (\Psi \vee \alpha'_1 \leqslant \alpha_1) \in \Phi \wedge \ldots \wedge (\Psi \vee \alpha'_n \leqslant \alpha_n) \in \Phi$

- $\forall \Psi, \alpha_1, \ldots, \alpha_n, \mathtt{t}, \alpha'_1, \ldots, \alpha'_n$.
  $(\Psi \vee (\alpha_1, \ldots, \alpha_n)\ \mathtt{t} \not\leqslant (\alpha'_1, \ldots, \alpha'_n)\ \mathtt{t}) \in \Phi \Rightarrow$
  $\wedge\ (\Psi \vee \alpha_1 \not\leqslant \alpha'_1 \vee \ldots \vee \alpha_n \not\leqslant \alpha'_n \vee \alpha'_1 \not\leqslant \alpha_1 \vee \ldots \vee \alpha'_n \not\leqslant \alpha_n) \in \Phi$

▶ *En cas d'invalidité d'un membre d'une disjonction, le reste doit être présent dans* Φ
(cf. SNotLeqSameVar, SLeqDiffParamed, SNotLeqSameExist*) :*

- $\forall \Psi, \alpha$.
  $(\Psi \vee \alpha \not\leqslant \alpha) \in \Phi \Rightarrow$
  $\wedge\ \Psi \in \Phi$

- $\forall \Psi, \alpha_1, \ldots, \alpha_n, \mathtt{t}, \alpha'_1, \ldots, \alpha'_p, \mathtt{u}$.
  $(\Psi \vee (\alpha_1, \ldots, \alpha_n)\ \mathtt{t} \leqslant (\alpha'_1, \ldots, \alpha'_p)\ \mathtt{u}) \in \Phi \Rightarrow$
  $\wedge\ \Psi \in \Phi$

- $\forall \Psi, \alpha_0, \Phi_0, \alpha^l, \alpha^r, \alpha_1, \ldots, \alpha_m$.
  $(\Psi \vee \epsilon(\alpha_0, \Phi_0, [\alpha^l; \alpha^r], \{\alpha_1, \ldots, \alpha_m\}) \not\leqslant \epsilon(\alpha_0, \Phi_0, [\alpha^l; \alpha^r], \{\alpha_1, \ldots, \alpha_m\})) \in \Phi \Rightarrow$
  $\wedge\ \Psi \in \Phi$

▶ *Saturation de la transitivité à travers un* α
(cf. SVarLeqParamed, SParamedLeqVar, SVarLeqVar, SVarNotLeqParamed, SParamedNotLeqVar, SVarNotLeqVar*) :*

- $\forall \Psi_1, \tau^l, \alpha, \Psi_2, \tau^r$.
  $(\Psi_1 \vee \tau^l \leqslant \alpha) \in \Phi \wedge (\Psi_2 \vee \alpha \leqslant \tau^r) \in \Phi \Rightarrow$
  $\wedge\ (\Psi_1 \vee \Psi_2 \vee \tau^l \leqslant \tau^r) \in \Phi$

- $\forall \Psi_1, \alpha, \tau^l, \Psi_2, \tau^r$.
  $(\Psi_1 \vee \alpha \leqslant \tau^r) \in \Phi \wedge (\Psi_2 \vee \alpha \not\leqslant \tau^l) \in \Phi \Rightarrow$
  $\wedge\ (\Psi_1 \vee \Psi_2 \vee \tau^r \not\leqslant \tau^l) \in \Phi$

- $\forall \Psi_1, \tau^l, \alpha, \Psi_2, \tau^r$.
  $(\Psi_1 \vee \tau^l \leqslant \alpha) \in \Phi \wedge (\Psi_2 \vee \tau^r \not\leqslant \alpha) \in \Phi \Rightarrow$
  $\wedge\ (\Psi_1 \vee \Psi_2 \vee \tau^r \not\leqslant \tau^l) \in \Phi$

► *Comparaison entre un type existentiel et une variable universelle dans sa portée*
(cf. SVarLeqExistIn, SExistLeqVarIn, SVarNotLeqExistIn, SExistNotLeqVarIn*) :*

- $\forall \Psi_1, \alpha_0, \Phi_0, \alpha^l, \alpha^r, \alpha_1, \ldots, \alpha_m, \alpha, \Psi_2, \tau^r \ .$
  $(\Psi_1 \vee \epsilon(\alpha_0, \Phi_0, [\alpha^l; \alpha^r], \{\alpha_1, \ldots, \alpha_m\}) \leqslant \alpha) \in \Phi \ \wedge \ \alpha \in [\alpha^l; \alpha^r] \ \wedge \ (\Psi_2 \vee \alpha \leqslant \tau^r) \in \Phi \ \Rightarrow$
  $(\Psi_1 \vee \Psi_2 \vee \epsilon(\alpha_0, \Phi_0, [\alpha^l; \alpha^r], \{\alpha_1, \ldots, \alpha_m\}) \leqslant \tau^r) \in \Phi$

- $\forall \Psi_1, \alpha, \alpha_0, \Phi_0, \alpha^l, \alpha^r, \alpha_1, \ldots, \alpha_m, \Psi_2, \tau^l \ .$
  $(\Psi_1 \vee \alpha \leqslant \epsilon(\alpha_0, \Phi_0, [\alpha^l; \alpha^r], \{\alpha_1, \ldots, \alpha_m\})) \in \Phi \ \wedge \ \alpha \in [\alpha^l; \alpha^r] \ \wedge \ (\Psi_2 \vee \tau^l \leqslant \alpha) \in \Phi \ \Rightarrow$
  $(\Psi_1 \vee \Psi_2 \vee \tau^l \leqslant \epsilon(\alpha_0, \Phi_0, [\alpha^l; \alpha^r], \{\alpha_1, \ldots, \alpha_m\})) \in \Phi$

- $\forall \Psi_1, \alpha_0, \Phi_0, \alpha^l, \alpha^r, \alpha_1, \ldots, \alpha_m, \alpha, \Psi_2, \tau^r \ .$
  $(\Psi_1 \vee \epsilon(\alpha_0, \Phi_0, [\alpha^l; \alpha^r], \{\alpha_1, \ldots, \alpha_m\}) \leqslant \alpha) \in \Phi \ \wedge \ \alpha \in [\alpha^l; \alpha^r] \ \wedge \ (\Psi_2 \vee \tau^r \not\leqslant \alpha) \in \Phi \ \Rightarrow$
  $(\Psi_1 \vee \Psi_2 \vee \tau^r \not\leqslant \epsilon(\alpha_0, \Phi_0, [\alpha^l; \alpha^r], \{\alpha_1, \ldots, \alpha_m\})) \in \Phi$

- $\forall \Psi_1, \alpha, \alpha_0, \Phi_0, \alpha^l, \alpha^r, \alpha_1, \ldots, \alpha_m, \Psi_2, \tau^l \ .$
  $(\Psi_1 \vee \alpha \leqslant \epsilon(\alpha_0, \Phi_0, [\alpha^l; \alpha^r], \{\alpha_1, \ldots, \alpha_m\})) \in \Phi \ \wedge \ \alpha \in [\alpha^l; \alpha^r] \ \wedge \ (\Psi_2 \vee \alpha \not\leqslant \tau^l) \in \Phi \ \Rightarrow$
  $(\Psi_1 \vee \Psi_2 \vee \epsilon(\alpha_0, \Phi_0, [\alpha^l; \alpha^r], \{\alpha_1, \ldots, \alpha_m\}) \not\leqslant \tau^l) \in \Phi$

- $\forall \Psi_1, \alpha_0, \Phi_0, \alpha^l, \alpha^r, \alpha_1, \ldots, \alpha_m, \alpha, \Psi_2, \tau^l \ .$
  $(\Psi_1 \vee \epsilon(\alpha_0, \Phi_0, [\alpha^l; \alpha^r], \{\alpha_1, \ldots, \alpha_m\}) \not\leqslant \alpha) \in \Phi \ \wedge \ \alpha \in [\alpha^l; \alpha^r] \ \wedge \ (\Psi_2 \vee \tau^l \leqslant \alpha) \in \Phi \ \Rightarrow$
  $(\Psi_1 \vee \Psi_2 \vee \epsilon(\alpha_0, \Phi_0, [\alpha^l; \alpha^r], \{\alpha_1, \ldots, \alpha_m\}) \not\leqslant \tau^l) \in \Phi$

- $\forall \Psi_1, \alpha, \alpha_0, \Phi_0, \alpha^l, \alpha^r, \alpha_1, \ldots, \alpha_m, \Psi_2, \tau^r \ .$
  $(\Psi_1 \vee \alpha \not\leqslant \epsilon(\alpha_0, \Phi_0, [\alpha^l; \alpha^r], \{\alpha_1, \ldots, \alpha_m\})) \in \Phi \ \wedge \ \alpha \in [\alpha^l; \alpha^r] \ \wedge \ (\Psi_2 \vee \alpha \leqslant \tau^r) \in \Phi \ \Rightarrow$
  $(\Psi_1 \vee \Psi_2 \vee \tau^r \not\leqslant \epsilon(\alpha_0, \Phi_0, [\alpha^l; \alpha^r], \{\alpha_1, \ldots, \alpha_m\})) \in \Phi$

► *Comparaison entre un type existentiel et une variable universelle hors de sa portée*
(cf. SVarLeqExistOut, SExistLeqVarOut, SVarNotLeqExistOut,
SExistNotLeqVarOut*) :*

- $\forall \Psi, \alpha_0, \Phi_0, \alpha^l, \alpha^r, \alpha_1, \ldots, \alpha_m, \alpha, \Psi_1, \ldots, \Psi_n, \tau_1^r, \ldots, \tau_n^r \ .$
  $(\Psi \vee \epsilon(\alpha_0, \Phi_0, [\alpha^l; \alpha^r], \{\alpha_1, \ldots, \alpha_m\}) \leqslant \alpha) \ \in \ \Phi \ \wedge \ \alpha \ \notin \ [\alpha^l; \alpha^r] \ \wedge$
  $\{ \ (\Psi_i, \tau_i^r) \ \}_{i=1}^n = \notin \textsc{rights}(\alpha_0, \Phi_0, \{\alpha_1, \ldots, \alpha_m\}) \ \Rightarrow$
  $(\Psi \vee (\overline{\Psi_1} \wedge \tau_1^r \leqslant \alpha) \vee \cdots \vee (\overline{\Psi_n} \wedge \tau_n^r \leqslant \alpha)) \ \in \ \Phi$

- $\forall \Psi, \alpha, \alpha_0, \Phi_0, \alpha^l, \alpha^r, \alpha_1, \ldots, \alpha_m, \Psi_1, \ldots, \Psi_n, \tau_1^l, \ldots, \tau_n^l \ .$
  $(\Psi \vee \alpha \leqslant \epsilon(\alpha_0, \Phi_0, [\alpha^l; \alpha^r], \{\alpha_1, \ldots, \alpha_m\})) \ \in \ \Phi \ \wedge \ \alpha \ \notin \ [\alpha^l; \alpha^r] \ \wedge$
  $\{ \ (\Psi_i, \tau_i^l) \ \}_{i=1}^n = \notin \textsc{lefts}(\alpha_0, \Phi_0, \{\alpha_1, \ldots, \alpha_m\}) \ \Rightarrow$
  $(\Psi \vee (\overline{\Psi_1} \wedge \alpha \leqslant \tau_1^l) \vee \cdots \vee (\overline{\Psi_n} \wedge \alpha \leqslant \tau_n^l)) \ \in \ \Phi$

- $\forall \Psi, \alpha_0, \Phi_0, \alpha^l, \alpha^r, \alpha_1, \ldots, \alpha_m, \alpha, \Psi_1, \ldots, \Psi_n, \tau_1^r, \ldots, \tau_n^r, \Psi'_1, \ldots, \Psi'_p, \tau_1'^r, \ldots, \tau_p'^r$.
  $(\Psi \vee \epsilon(\alpha_0, \Phi_0, [\alpha^l; \alpha^r], \{\alpha_1, \ldots, \alpha_m\}) \leqslant \alpha) \in \Phi \wedge \alpha \notin [\alpha^l; \alpha^r] \wedge$
  $\{ (\Psi_i, \tau_i^r) \}_{i=1}^n = \notin \textsc{rights}(\alpha_0, \Phi_0, \{\alpha_1, \ldots, \alpha_m\}) \wedge$
  $\{ (\Psi'_i, \tau_i'^r) \}_{i=1}^p = \overline{\notin \textsc{lefts}}(\alpha_0, \Phi_0, \{\alpha_1, \ldots, \alpha_m\}) \Rightarrow$
  $(\Psi \vee (\overline{\Psi_1} \wedge \alpha \not\leqslant \tau_1^r) \vee \cdots \vee (\overline{\Psi_n} \wedge \alpha \not\leqslant \tau_n^r) \vee (\overline{\Psi'_1} \wedge \tau_1'^r \leqslant \alpha) \vee \cdots \vee (\overline{\Psi'_p} \wedge \tau_p'^r \leqslant \alpha)) \in \Phi$

- $\forall \Psi, \alpha, \alpha_0, \Phi_0, \alpha^l, \alpha^r, \alpha_1, \ldots, \alpha_m, \Psi_1, \ldots, \Psi_n, \tau_1^l, \ldots, \tau_n^l, \Psi'_1, \ldots, \Psi'_p, \tau_1'^l, \ldots, \tau_p'^l$.
  $(\Psi \vee \alpha \leqslant \epsilon(\alpha_0, \Phi_0, [\alpha^l; \alpha^r], \{\alpha_1, \ldots, \alpha_m\})) \in \Phi \wedge \alpha \notin [\alpha^l; \alpha^r] \wedge$
  $\{ (\Psi_i, \tau_i^l) \}_{i=1}^n = \notin \textsc{lefts}(\alpha_0, \Phi_0, \{\alpha_1, \ldots, \alpha_m\}) \wedge$
  $\{ (\Psi'_i, \tau_i'^l) \}_{i=1}^p = \overline{\notin \textsc{rights}}(\alpha_0, \Phi_0, \{\alpha_1, \ldots, \alpha_m\}) \Rightarrow$
  $(\Psi \vee (\overline{\Psi_1} \wedge \tau_1^l \not\leqslant \alpha) \vee \cdots \vee (\overline{\Psi_n} \wedge \tau_n^l \not\leqslant \alpha) \vee (\overline{\Psi'_1} \wedge \alpha \leqslant \tau_1'^l) \vee \cdots \vee (\overline{\Psi'_p} \wedge \alpha \leqslant \tau_p'^l)) \in \Phi$

▶ *Comparaison entre un type existentiel et un type paramétré*
(cf. SParamedLeqExist, SExistLeqParamed, SParamedNotLeqExist, SExistNotLeqParamed*) :*

- $\forall \Psi, \alpha_0, \Phi_0, \alpha^l, \alpha^r, \alpha_1, \ldots, \alpha_m, \alpha'_1, \ldots, \alpha'_q, \mathtt{t}, \Psi_1, \ldots, \Psi_n, \tau_1^r, \ldots, \tau_n^r$.
  $(\Psi \vee \epsilon(\alpha_0, \Phi_0, [\alpha^l; \alpha^r], \{\alpha_1, \ldots, \alpha_m\}) \leqslant (\alpha'_1, \ldots, \alpha'_q)\ \mathtt{t}) \in \Phi \wedge$
  $\{ (\Psi_i, \tau_i^r) \}_{i=1}^n = \notin \textsc{rights}(\alpha_0, \Phi_0, \{\alpha_1, \ldots, \alpha_m\}) \Rightarrow$
  $(\Psi \vee (\overline{\Psi_1} \wedge \tau_1^r \leqslant (\alpha'_1, \ldots, \alpha'_q)\ \mathtt{t}) \vee \cdots \vee (\overline{\Psi_n} \wedge \tau_n^r \leqslant (\alpha'_1, \ldots, \alpha'_q)\ \mathtt{t})) \in \Phi$

- $\forall \Psi, \alpha'_1, \ldots, \alpha'_q, \mathtt{t}, \alpha_0, \Phi_0, \alpha^l, \alpha^r, \alpha_1, \ldots, \alpha_m, \Psi_1, \ldots, \Psi_n, \tau_1^l, \ldots, \tau_n^l$.
  $(\Psi \vee (\alpha'_1, \ldots, \alpha'_q)\ \mathtt{t} \leqslant \epsilon(\alpha_0, \Phi_0, [\alpha^l; \alpha^r], \{\alpha_1, \ldots, \alpha_m\})) \in \Phi \wedge$
  $\{ (\Psi_i, \tau_i^l) \}_{i=1}^n = \notin \textsc{lefts}(\alpha_0, \Phi_0, \{\alpha_1, \ldots, \alpha_m\}) \Rightarrow$
  $(\Psi \vee (\overline{\Psi_1} \wedge (\alpha'_1, \ldots, \alpha'_q)\ \mathtt{t} \leqslant \tau_1^l) \vee \cdots \vee (\overline{\Psi_n} \wedge (\alpha'_1, \ldots, \alpha'_q)\ \mathtt{t} \leqslant \tau_n^l)) \in \Phi$

- $\forall \Psi, \alpha_0, \Phi_0, \alpha^l, \alpha^r, \alpha_1, \ldots, \alpha_m, \alpha'_1, \ldots, \alpha'_q, \mathtt{t}, \Psi_1, \ldots, \Psi_n, \tau_1^r, \ldots, \tau_n^r, \Psi'_1, \ldots, \Psi'_p, \tau_1'^r, \ldots, \tau_p'^r$.
  $(\Psi \vee \epsilon(\alpha_0, \Phi_0, [\alpha^l; \alpha^r], \{\alpha_1, \ldots, \alpha_m\}) \leqslant (\alpha'_1, \ldots, \alpha'_q)\ \mathtt{t}) \in \Phi \wedge$
  $\{ (\Psi_i, \tau_i^r) \}_{i=1}^n = \notin \textsc{rights}(\alpha_0, \Phi_0, \{\alpha_1, \ldots, \alpha_m\}) \wedge$
  $\{ (\Psi'_i, \tau_i'^r) \}_{i=1}^p = \overline{\notin \textsc{lefts}}(\alpha_0, \Phi_0, \{\alpha_1, \ldots, \alpha_m\}) \Rightarrow$
  $(\Psi \vee (\overline{\Psi_1} \wedge (\alpha'_1, \ldots, \alpha'_q)\ \mathtt{t} \not\leqslant \tau_1^r) \vee \cdots \vee (\overline{\Psi_n} \wedge (\alpha'_1, \ldots, \alpha'_q)\ \mathtt{t} \not\leqslant \tau_n^r) \vee$
  $(\overline{\Psi'_1} \wedge \tau_1'^r \leqslant (\alpha'_1, \ldots, \alpha'_q)\ \mathtt{t}) \vee \cdots \vee (\overline{\Psi'_p} \wedge \tau_p'^r \leqslant (\alpha'_1, \ldots, \alpha'_q)\ \mathtt{t})) \in \Phi$

- $\forall \Psi, \alpha'_1, \ldots, \alpha'_q, \mathtt{t}, \alpha_0, \Phi_0, \alpha^l, \alpha^r, \alpha_1, \ldots, \alpha_m, \Psi_1, \ldots, \Psi_n, \tau_1^l, \ldots, \tau_n^l, \Psi'_1, \ldots, \Psi'_p, \tau_1'^l, \ldots, \tau_p'^l$.
  $(\Psi \vee (\alpha'_1, \ldots, \alpha'_q)\ \mathtt{t} \leqslant \epsilon(\alpha_0, \Phi_0, [\alpha^l; \alpha^r], \{\alpha_1, \ldots, \alpha_m\})) \in \Phi \wedge$
  $\{ (\Psi_i, \tau_i^l) \}_{i=1}^n = \notin \textsc{lefts}(\alpha_0, \Phi_0, \{\alpha_1, \ldots, \alpha_m\}) \wedge$
  $\{ (\Psi'_i, \tau_i'^l) \}_{i=1}^p = \overline{\notin \textsc{rights}}(\alpha_0, \Phi_0, \{\alpha_1, \ldots, \alpha_m\}) \Rightarrow$
  $(\Psi \vee (\overline{\Psi_1} \wedge \tau_1^l \not\leqslant (\alpha'_1, \ldots, \alpha'_q)\ \mathtt{t}) \vee \cdots \vee (\overline{\Psi_n} \wedge \tau_n^l \not\leqslant (\alpha'_1, \ldots, \alpha'_q)\ \mathtt{t}) \vee$
  $(\overline{\Psi'_1} \wedge (\alpha'_1, \ldots, \alpha'_q)\ \mathtt{t} \leqslant \tau_1'^l) \vee \cdots \vee (\overline{\Psi'_p} \wedge (\alpha'_1, \ldots, \alpha'_q)\ \mathtt{t} \leqslant \tau_p'^l)) \in \Phi$

▶ *Comparaison entre deux types existentiels différents*
(cf. SExistLeqExist, SExistNotLeqExist*) :*

- $\forall \Psi, \alpha_0, \Phi_0, \alpha^l, \alpha^r, \alpha_1, \ldots, \alpha_m, \alpha'_0, \Phi'_0, \alpha'_l, \alpha'_r, \alpha'_1, \ldots, \alpha'_{m'}$.
  $(\Psi \vee \epsilon(\alpha_0, \Phi_0, [\alpha^l; \alpha^r], \{\alpha_1, \ldots, \alpha_m\}) \leqslant \epsilon(\alpha'_0, \Phi'_0, [\alpha'_l; \alpha'_r], \{\alpha'_1, \ldots, \alpha'_{m'}\})) \in \Phi \ \wedge$
  $\{ (\Psi_i, \tau^r_i) \}^n_{i=1} = \textsc{rights}(\alpha_0, \Phi_0) \ \wedge \ \{ (\Psi'_i, \tau'^l_i) \}^p_{i=1} = \textsc{lefts}(\alpha'_0, \Phi'_0) \ \Rightarrow$
  $\wedge \ (\Psi \ \vee \ (\overline{\Psi_1} \wedge \overline{\Psi'_1} \wedge \tau^r_1 \leqslant \tau'^l_1) \vee \cdots \vee (\overline{\Psi_1} \wedge \overline{\Psi'_p} \wedge \tau^r_1 \leqslant \tau'^l_p)$
  $\vdots \qquad\qquad \vdots$
  $\vee \ (\overline{\Psi_n} \wedge \overline{\Psi'_1} \wedge \tau^r_n \leqslant \tau'^l_1) \vee \cdots \vee (\overline{\Psi_n} \wedge \overline{\Psi'_p} \wedge \tau^r_n \leqslant \tau'^l_p)) \ \in \ \Phi$

- $\forall \Psi, \alpha_0, \Phi_0, \alpha^l, \alpha^r, \alpha_1, \ldots, \alpha_m, \alpha'_0, \Phi'_0, \alpha'_l, \alpha'_r, \alpha'_1, \ldots, \alpha'_{m'}, \Psi^l_1, \ldots, \Psi^l_n, \tau^l_1, \ldots, \tau^l_n, \Psi'^l_1, \ldots, \Psi'^l_p,$
  $\tau'^l_1, \ldots, \tau'^l_p, \Psi^r_1, \ldots, \Psi^r_q, \tau^r_1, \ldots, \tau^r_q, \Psi'^r_1, \ldots, \Psi'^r_s, \tau'^r_1, \ldots, \tau'^r_s$.
  $(\Psi \vee \epsilon(\alpha_0, \Phi_0, [\alpha^l; \alpha^r], \{\alpha_1, \ldots, \alpha_m\}) \not\leqslant \epsilon(\alpha'_0, \Phi'_0, [\alpha'_l; \alpha'_r], \{\alpha'_1, \ldots, \alpha'_{m'}\})) \in \Phi \ \wedge$
  $\{ (\Psi^l_i, \tau^l_i) \}^n_{i=1} = \textsc{lefts}(\alpha_0, \Phi_0) \ \wedge \ \{ (\Psi'^l_i, \tau'^l_i) \}^p_{i=1} = \overline{\textsc{rights}}(\alpha'_0, \Phi'_0) \ \wedge$
  $\{ (\Psi^r_i, \tau^r_i) \}^q_{i=1} = \textsc{rights}(\alpha_0, \Phi_0) \ \wedge \ \{ (\Psi'^r_i, \tau'^r_i) \}^s_{i=1} = \overline{\textsc{lefts}}(\alpha'_0, \Phi'_0) \ \Rightarrow$
  $\wedge \ (\Psi \ \vee \ (\overline{\Psi^l_1} \wedge \overline{\Psi^r_1} \wedge \tau^l_1 \leqslant \tau^r_1) \vee \cdots \vee (\overline{\Psi^l_1} \wedge \overline{\Psi^r_q} \wedge \tau^l_1 \leqslant \tau^r_q)$
  $\vdots \qquad\qquad \vdots$
  $\vee \ (\overline{\Psi^l_n} \wedge \overline{\Psi^r_1} \wedge \tau^l_n \leqslant \tau^r_1) \vee \cdots \vee (\overline{\Psi^l_n} \wedge \overline{\Psi^r_q} \wedge \tau^l_n \leqslant \tau^r_q)$
  $\vee \ (\overline{\Psi'^l_1} \wedge \overline{\Psi^r_1} \wedge \tau'^l_1 \not\leqslant \tau^r_1) \vee \cdots \vee (\overline{\Psi'^l_1} \wedge \overline{\Psi^r_q} \wedge \tau'^l_1 \not\leqslant \tau^r_q)$
  $\vdots \qquad\qquad \vdots$
  $\vee \ (\overline{\Psi'^l_p} \wedge \overline{\Psi^r_1} \wedge \tau'^l_p \not\leqslant \tau^r_1) \vee \cdots \vee (\overline{\Psi'^l_p} \wedge \overline{\Psi^r_q} \wedge \tau'^l_p \not\leqslant \tau^r_q)$
  $\vee \ (\overline{\Psi^l_1} \wedge \overline{\Psi'^r_1} \wedge \tau^l_1 \not\leqslant \tau'^r_1) \vee \cdots \vee (\overline{\Psi^l_1} \wedge \overline{\Psi'^r_s} \wedge \tau^l_1 \not\leqslant \tau'^r_s)$
  $\vdots \qquad\qquad \vdots$
  $\vee \ (\overline{\Psi^l_n} \wedge \overline{\Psi'^r_1} \wedge \tau^l_n \not\leqslant \tau'^r_1) \vee \cdots \vee (\overline{\Psi^l_n} \wedge \overline{\Psi'^r_s} \wedge \tau^l_n \not\leqslant \tau'^r_s)) \in \Phi$

5.5.2 Mise à jour des lemmes

Comme chaque fois, les deux lemmes, dont la démonstration doit être adaptée pour prendre en compte les nouvelles relations et constructions de type, sont les lemmes de réduction du sujet et le lemme spécifiant que les expressions bloquées sont non-typables. Nous ne donnons ici que les points à modifier dans les démonstrations en partant de celles du chapitre 3. Ces points concernent principalement le filtrage de motifs.

Lemme 5 (Réduction du sujet par $(\longrightarrow)$)**.**

Soient :

- e_1 *et* e_2 *deux expressions vérifiant* $e_1 \longrightarrow e_2$
- α *une variable de type*
- Φ *un ensemble de contraintes valide et saturé*

- Φ'_1 *l'ensemble de contraintes obtenu par typage de* $\mathtt{e}_1 : \alpha$ *dans* $(\Phi, \varnothing, \varnothing)$ *en dérivant l'arbre d'inférence au dessus de* $\Phi, \varnothing \vdash \varnothing \vee \mathtt{e}_1 : \alpha \rhd \Phi'_1$.

Alors le typage de $\mathtt{e}_2 : \alpha$ *dans* $(\Phi, \varnothing, \varnothing)$ *réussit et l'ensemble de contraintes* Φ'_2 *obtenu en dérivant l'arbre d'inférence au dessus de* $\Phi, \varnothing \vdash \varnothing \vee \mathtt{e}_2 : \alpha \rhd \Phi'_2$ *vérifie* $\Phi'_2 \leqslant \Phi'_1$.

Dans ce lemme, tout comme dans son équivalent du chapitre 3, le Ψ est considéré vide, comme l'est Γ. Il ne serait pas utile de généraliser ce théorème à un Ψ non-vide puisque le contexte d'évaluation ne nous autorise jamais à réduire de sous-expression sous un filtrage.

Démonstration (Lemme 5) :

Comme habituellement, on distingue différents cas en fonction de la structure syntaxique de $\mathtt{e}_1$ (déterminant la règle de réduction par $(\longrightarrow)$ qui lui est appliquée), et on montre qu'il est possible de construire l'arbre d'inférence au dessus de $\Phi, \varnothing \vdash \varnothing \vee \mathtt{e}_2 : \alpha \rhd \Phi'_2$ à partir de l'arbre d'inférence au dessus de $\Phi, \varnothing \vdash \varnothing \vee \mathtt{e}_1 : \alpha \rhd \Phi'_1$ qui est constructible par hypothèse. Le seul cas intéressant pour cette preuve concerne la réduction d'un filtrage de motifs. La preuve de ce lemme dans les autres cas est similaire à celle du chapitre 3.

- Cas « $\mathtt{match\ K(v_1,\ldots,v_n)\ with\ \ldots \parallel K(x_1,\ldots,x_n) \to e \parallel \ldots} \longrightarrow \mathtt{e[x_i \mapsto v_i]}_{i=1}^{n}$ » : L'expression $\mathtt{e}_1$ est typable, le constructeur $\mathtt{K}$ provient donc de la définition d'un type **GADT** de la forme :

$$\mathtt{type}\ (\alpha_1,\ldots,\alpha_p)\ \mathtt{t} = \ldots \mid \exists \beta_1,\ldots,\beta_q\ .\ \mathtt{K}(\beta_1,\ldots,\beta_n)\ [\ \Phi_0\ ] \mid \ldots$$

L'arbre d'inférence au dessus de :

$$\Phi, \varnothing, \vdash \varnothing \vee \mathtt{match\ K}\ (\mathtt{v}_1,\ldots,\mathtt{v}_n)\ \mathtt{with}\ \ldots \parallel \mathtt{K}\ (\mathtt{x}_1,\ldots,\mathtt{x}_n) \to \mathtt{e} \parallel \ldots : \alpha \rhd \Phi'_1$$

est obligatoirement de la forme suivante :

$$\textsc{TMatch}\ \dfrac{\dfrac{\ldots}{\ldots} \quad \textsc{TCase}\ \dfrac{\overset{T_3}{\triangledown}}{\Phi'', \varnothing \vdash \varnothing \vee \mathtt{K}\ (\mathtt{x}_1,\ldots,\mathtt{x}_n) \to \mathtt{e} : \alpha_e, \alpha'_1,\ldots,\alpha'_p \rhd \Phi'_1} \quad \dfrac{\ldots}{\ldots}}{\Phi, \varnothing, \vdash \varnothing \vee \mathtt{match\ K}\ (\mathtt{v}_1,\ldots,\mathtt{v}_n)\ \mathtt{with}\ \ldots \parallel \mathtt{K}\ (\mathtt{x}_1,\ldots,\mathtt{x}_n) \to \mathtt{e} \parallel \ldots : \alpha \rhd \Phi'_1}$$

$$\textsc{TConstr}\ \dfrac{\overset{T_1}{\triangledown}}{\Phi, \varnothing \vdash \varnothing \vee \mathtt{K}\ (\mathtt{v}_1,\ldots,\mathtt{v}_n) : \alpha_e \rhd \Phi'} \qquad \dfrac{\overset{T_2}{\triangledown}}{\Phi', \varnothing \vdash \varnothing \vee \alpha_e \leqslant (\alpha'_1,\ldots,\alpha'_p)\ \mathtt{t} \rhd \Phi''}$$

dans lequel le sous-arbre T_1 est de la forme :

$$\dfrac{\overset{T_1^v}{\triangledown}}{\Phi_1, \varnothing \vdash \varnothing \vee v_1 : \beta_1'' \rhd \Phi_2} \quad \dfrac{\cdots}{\cdots} \quad \dfrac{\overset{T_n^v}{\triangledown}}{\Phi_n, \varnothing \vdash \varnothing \vee v_n : \beta_n'' \rhd \Phi_{n+1}}$$

$$\textsc{TConstr}\ \dfrac{\dfrac{\overset{T_4}{\triangledown}}{\Phi, \varnothing \vdash \Phi_0'' \rhd \Phi_1} \quad \dfrac{\overset{T_5}{\triangledown}}{\Phi_{n+1} \vdash (\alpha_1'', \ldots, \alpha_p'')\ \mathtt{t} \leqslant \alpha_e \rhd \Phi'}}{\Phi, \varnothing \vdash \varnothing \vee \mathtt{K}\ (\mathtt{v}_1, \ldots, \mathtt{v}_n) : \alpha_e \rhd \Phi'}$$

avec $\Phi_0'' = \Phi_0[\alpha_i \mapsto \alpha_i'']_{i=1}^p[\beta_i \mapsto \beta_i'']_{i=1}^q$

et le sous-arbre T_3 de la forme :

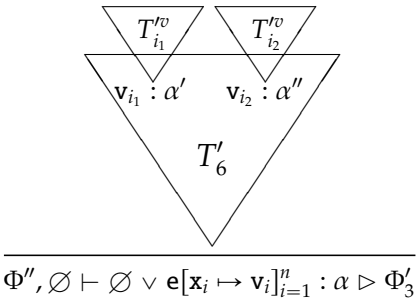
$$\dfrac{\overset{\mathtt{x}_{i_1} : \alpha' \quad \mathtt{x}_{i_2} : \alpha''}{T_6}}{\Phi'', \{\ x_i : \beta_i'\ \}_{i=1}^n \vdash \overline{\Phi_0'} \vee \mathtt{e} : \alpha_e \rhd \Phi_1''}$$

$$\dfrac{\overset{T_1^r}{\triangledown}}{\Phi_1'' \vdash \epsilon(\beta_1', \Phi_0', [\alpha^l; \alpha^r], \{\beta_1', ..., \beta_q'\}) \leqslant \beta_1' \rhd \Phi_2''} \quad \dfrac{\cdots}{\cdots} \quad \dfrac{\overset{T_q^r}{\triangledown}}{\Phi_q'' \vdash \epsilon(\beta_q', \Phi_0', [\alpha^l; \alpha^r], \{\beta_1', ..., \beta_q'\}) \leqslant \beta_q' \rhd \Phi_{q+1}''}$$

$$\textsc{TCase}\ \dfrac{\dfrac{\overset{T_1^l}{\triangledown}}{\Phi_{q+1}'' \vdash \beta_1' \leqslant \epsilon(\beta_1', \Phi_0', [\alpha^l; \alpha^r], \{\beta_1', ..., \beta_q'\}) \rhd \Phi_{q+2}''} \quad \dfrac{\cdots}{\cdots} \quad \dfrac{\overset{T_q^l}{\triangledown}}{\Phi_{2q}'' \vdash \beta_q' \leqslant \epsilon(\beta_q', \Phi_0', [\alpha^l; \alpha^r], \{\beta_1', ..., \beta_q'\}) \rhd \Phi_1'}}{\Phi'', \varnothing \vdash \varnothing \vee \mathtt{K}\ (\mathtt{x}_1, \ldots, \mathtt{x}_n) \to \mathtt{e} : \alpha_e, \alpha_1', \ldots, \alpha_p' \rhd \Phi_1'}$$

où $\Phi_0' = \Phi_0[\alpha_i \mapsto \alpha_i']_{i=1}^p[\beta_i \mapsto \beta_i']_{i=1}^q$ et $[\alpha^l; \alpha^r]$ représente l'intervalle de variables de type universelles générées lors du typage de $\mathtt{e}$ dans le sous-arbre T_6.

Nous commençons par montrer qu'il est possible de construire un arbre d'inférence au dessus de $\Phi'', \varnothing \vdash \varnothing \vee \mathtt{e}[\mathtt{x}_i \mapsto \mathtt{v}_i]_{i=1}^n : \alpha \rhd \Phi_3'$ en remplaçant dans T_6 les sous-arbres correspondant aux différentes occurrences des $\mathtt{x}_i$ dans l'expression $\mathtt{e}$ par une adaptation des sous-arbres T_i^v de T_1.



Néanmoins, pour assurer que la construction du nouvel arbre est possible et que la relation $\Phi'_2 \leqslant \Phi'_1$ est vérifiée, il faut en particulier montrer que ce remplacement des sous-arbres de T_6 n'impose pas de contrainte supplémentaire, c'est-à-dire de contraintes qui n'étaient pas déjà présentes dans Φ'_1.

Suppression du $\overline{\Phi'_0}$

Le $\overline{\Phi'_0}$ présent dans la racine de T_6 est absent de la racine de l'arbre que l'on souhaite montrer constructible. Il n'est a priori pas évident que supprimer cette disjonction de contraintes alternatives dans le nouvel arbre n'engendre pas de clash. Ainsi, il se pourrait que l'une des contraintes générées par le typage de `e` soit invalide mais réussisse à imposer au saturateur de contraintes de montrer que l'un des membres de $\overline{\Phi'_0}$ est valide et compatible avec les autres contraintes de Φ'_1. Nous allons montrer qu'un tel scénario est impossible.

Pour ce faire, nous allons démontrer que tous les membres de $\overline{\Phi'_0}$ clashent avec le reste de Φ'_1, et ainsi que toutes les contraintes générées lors du typage de `e`, qui se retrouvent dans des disjonctions contenant $\overline{\Phi'_0}$, engendrent après saturation les mêmes disjonctions sans le $\overline{\Phi'_0}$.

Nous commençons par remarquer que le sous-arbre T_4 a ajouté $\Phi_0[\alpha_i \mapsto \alpha''_i]_{i=1}^{p}[\beta_i \mapsto \beta''_i]_{i=1}^{q}$ dans l'ensemble de contraintes, qui diffère de Φ'_0 uniquement par le renommage des variables $\{\ \alpha_i\ \}_{i=1}^{p}$ et $\{\ \beta_i\ \}_{i=1}^{q}$.

Les sous-arbres T_5 et T_2 ont ajouté respectivement les contraintes $(\alpha'_1, \ldots, \alpha'_p)\ \mathtt{t} \leqslant \alpha_e$ et $\alpha_e \leqslant (\alpha''_1, \ldots, \alpha''_p)\ \mathtt{t}$, ce qui, après saturation, a engendré les contraintes $(\alpha'_i \leqslant \alpha''_i)$ et $(\alpha''_i \leqslant \alpha'_i)$ pour tout $i \in [1; p]$. Ainsi, pour tout $i \in [1; p]$, toute contrainte, de sous-typage comme de non-sous-typage, imposée sur α'_i est, après saturation, imposée sur α''_i, et réciproquement.

Analysons maintenant les différentes formes possibles des contraintes de $\overline{\Phi'_0}$ et montrons qu'elles sont incompatibles avec les autres contraintes de Φ'_1 :

- Si une contrainte de $\overline{\Phi'_0}$ est de la forme $\alpha'_i \leqslant \alpha'_j$ (resp. $\alpha'_i \not\leqslant \alpha'_j$) pour $i \in [1; p]$ et $j \in [1; p]$, son équivalent $\alpha''_i \leqslant \alpha''_j$ (resp. $\alpha''_i \not\leqslant \alpha''_j$) a été imposé par saturation, et est entré en collision avec la contrainte inverse provenant de Φ''_0.
- La situation est similaire si la contrainte de $\overline{\Phi'_0}$ considérée est de la forme $(\alpha'_i \leqslant (\ldots)\ \mathtt{t})$

(resp. $(\alpha'_i \not\leqslant (\dots)\ \mathtt{t})$, $((\dots)\ \mathtt{t} \leqslant \alpha'_i)$ ou $((\dots)\ \mathtt{t} \not\leqslant \alpha'_i)$), après saturation, son équivalent $(\alpha''_i \leqslant (\dots)\ \mathtt{t})$ (resp. $(\alpha''_i \not\leqslant (\dots)\ \mathtt{t})$, $((\dots)\ \mathtt{t} \leqslant \alpha''_i)$ ou $((\dots)\ \mathtt{t} \not\leqslant \alpha''_i)$) est entré en collision avec la contrainte inverse provenant de Φ''_0.

Remarque : comme nous allons le voir maintenant, si la contrainte provenant de $\overline{\Phi'_0}$ fait intervenir un $\beta'_{i'}$, la situation est différente car les β'_i ne sont pas liés aux β''_i par une double inégalité. En revanche, ils sont liés par une double inégalité à des types existentiels ($\epsilon(...)$) grâce aux sous-arbres T^r_i et T^l_i pour $i \in [1;q]$.

- Si une contrainte est de la forme $\beta'_i \leqslant (\alpha'''_1, \dots, \alpha'''_k)\ \mathtt{t}$, grâce à la contrainte $(\epsilon(\beta'_i, \Phi'_0, [\alpha^l; \alpha^r], \{\beta'_1, \dots, \beta'_q\}) \leqslant \beta'_i)$ ajoutée dans le sous-arbre T^r_i, la contrainte $\epsilon(\beta'_i, \Phi'_0, [\alpha^l; \alpha^r], \{\beta'_1, \dots, \beta'_q\}) \leqslant (\alpha'''_1, \dots, \alpha'''_k)\ \mathtt{t}$ a été obtenue après saturation. La règle SExistLeqParamed a alors extrait de Φ'_0 tous les τ^r qui ne sont pas des β'_i et apparaissant dans Φ'_0 dans une contrainte de la forme $\beta'_i \leqslant \tau^r$. S'il n'y en avait aucun, une disjonction vide a été générée, ce qui a engendré un clash. S'il y en avait un ou plusieurs, une disjonction dont tous les membres étaient de la forme $\tau^r \leqslant (\alpha'''_1, \dots, \alpha'''_k)\ \mathtt{t}$ a été générée. Il faut alors montrer que chaque membre de cette disjonction est soit invalide, soit a engendré un clash grâce aux autres contraintes de Φ'_1. Comme nous l'avons dit, les τ^r ne peuvent pas être des β_i. Les trois formes possibles pour un τ^r sont :
  - ◇ Une variable universelle α'_j. La contrainte $(\alpha'_j \leqslant (\alpha'''_1, \dots, \alpha'''_k)\ \mathtt{t})$ a alors été générée, et comme la contrainte $(\alpha''_j \leqslant \alpha'_j)$ est également présente dans Φ'_1, nous en déduisons que $(\alpha''_j \leqslant (\alpha'''_1, \dots, \alpha'''_k)\ \mathtt{t})$ aussi. Cependant, comme Φ'_0 contient la contrainte $(\beta'_i \leqslant \alpha'_j)$, nous en déduisons que Φ''_0 contient $(\beta''_i \leqslant \alpha''_j)$ et donc que Φ'_1 contient $(\beta''_i \leqslant (\alpha'''_1, \dots, \alpha'''_k)\ \mathtt{t})$. Par ailleurs, comme $\overline{\Phi'_0}$ contient la contrainte $(\beta'_i \leqslant (\alpha'''_1, \dots, \alpha'''_k)\ \mathtt{t})$, alors Φ''_0 contient obligatoirement une contrainte de la forme $(\beta''_i \not\leqslant (\alpha''''_1, \dots, \alpha''''_k)\ \mathtt{t})$ vérifiant $\forall l \in [1;k]\ .\ \exists m \mid (\alpha'''_l = \alpha'_m \wedge \alpha''''_l = \alpha''_m) \vee (\alpha'''_l = \beta'_m \wedge \alpha''''_l = \beta''_m)$ et qui, avec la contrainte $(\beta''_i \leqslant (\alpha'''_1, \dots, \alpha'''_k)\ \mathtt{t})$, a engendré, via la règle SVarLeqParamed, la contrainte $(\alpha'''_1, \dots, \alpha'''_k)\ \mathtt{t} \not\leqslant (\alpha''''_1, \dots, \alpha''''_k)\ \mathtt{t}$. L'application de la règle SNotLeqSameParamed a alors généré la disjonction $\alpha'''_1 \not\leqslant \alpha''''_1 \vee \dots \vee \alpha'''_k \not\leqslant \alpha''''_k \vee \alpha''''_1 \not\leqslant \alpha'''_1 \vee \dots \vee \alpha''''_k \not\leqslant \alpha'''_k$. Il ne reste alors qu'à montrer que chaque membre de cette disjonction est incompatible avec les autres contraintes de Φ'_1. Pour chaque membre de cette disjonction, il n'y a que deux possibilités. Soit la relation de non-sous-typage relie un α'_l à un α''_l, et dans ce cas, cette relation est incompatible avec les relations de sous-typage reliant les α'_l à leur α''_l correspondant. Soit elle relie un β'_l à un β''_l, et dans ce cas, cette relation a été reportée par transitivité sur les types existentiels liés par double inégalité aux β'_l. Comme les β''_l sont en dehors de la portée des types existentiels associés aux β'_l, l'une des règles SVarLeqExistOut et SExistLeqVarOut a été appliquée et a généré une disjonction de contraintes sur β'_l extraite de Φ'_0 qui est obligatoirement incompatible avec la conjonction de contraintes inverse imposée sur β'_l dans T_1.
  - ◇ Un type paramétré $(\dots)\ \mathtt{u}$ avec $\mathtt{t} \neq \mathtt{u}$. La contrainte $(\dots)\ \mathtt{u} \leqslant (\dots)\ \mathtt{t}$ est invalide.

◇ Un type paramétré $(\alpha_1^4, \ldots, \alpha_k^4)$ t. La contrainte extraite de Φ'_0 est alors $(\alpha_1^4, \ldots, \alpha_k^4)$ t $\leqslant (\alpha_1''', \ldots, \alpha_k''')$ t. Sa saturation a engendré, via la règle SLeqSameParamed, les $2k$ contraintes de sous-typage $(\alpha_1^4 \leqslant \alpha_1''')$, …, $(\alpha_k^4 \leqslant \alpha_k''')$ et $(\alpha_1''' \leqslant \alpha_1^4)$, …, $(\alpha_k''' \leqslant \alpha_k^4)$.

Par ailleurs, comme $\overline{\Phi'_0}$ contient la contrainte $(\beta'_i \leqslant (\alpha_1''', \ldots, \alpha_k''')$ t$)$, alors Φ''_0 contient obligatoirement une contrainte de la forme $(\beta_i'' \not\leqslant (\alpha_1^5, \ldots, \alpha_k^5)$ t$)$ vérifiant $\forall l \in [1;k]\,.\,\exists m \mid (\alpha_l''' = \alpha'_m \wedge \alpha_l^5 = \alpha_m'') \vee (\alpha_l''' = \beta'_m \wedge \alpha_l^5 = \beta_m'')$. De plus, comme Φ'_0 contient la contrainte $(\beta'_i \leqslant (\alpha^4, \ldots, \alpha^4)$ t$)$, alors Φ''_0 contient obligatoirement une contrainte de la forme $(\beta_i'' \leqslant (\alpha^6, \ldots, \alpha^6)$ t$)$ vérifiant $\forall l \in [1;k]\,.\,\exists m \mid (\alpha_l^4 = \alpha'_m \wedge \alpha_l^6 = \alpha_m'') \vee (\alpha_l^4 = \beta'_m \wedge \alpha_l^6 = \beta_m'')$. La saturation a donc engendré dans Φ'_1 la contrainte $((\alpha_1^6, \ldots, \alpha_k^6)$ t $\not\leqslant (\alpha_1^5, \ldots, \alpha^5)$ t$)$, et par application de la règle SNotLeqSameParamed la disjonction de contraintes $(\alpha_1^6 \not\leqslant \alpha_1^5 \vee \cdots \vee \alpha_k^6 \not\leqslant \alpha_k^5 \vee \alpha_1^5 \not\leqslant \alpha_1^6 \vee \cdots \vee \alpha_k^5 \not\leqslant \alpha_k^6)$.

Il suffit alors de montrer que chaque membre de cette disjonction est incompatible avec les autres contraintes de Φ'_1. Montrons le pour $\alpha_1^6 \not\leqslant \alpha_1^5$, les autres cas sont très similaires. Il faut distinguer quatre cas, en fonction de la nature de α_1^6/α_1^4 et de α_1^5/α_1''' :

⊛ Si α_1^6/α_1^4 et α_1^5/α_1''' sont des α_m''/α'_m.
Dans ce cas, les variables α_1^6 et α_1^4 sont liées par une double inégalité, ainsi que α_1^5 et α_1'''. La contrainte $(\alpha_1^4 \leqslant \alpha_1''')$ est alors incompatible avec $(\alpha_1^6 \not\leqslant \alpha_1^5)$.

⊛ Si α_1^6/α_1^4 sont des α_m''/α'_m et α_1^5/α_1''' sont des β_m''/β'_m.
Dans ce cas, les variables α_1^6 et α_1^4 sont liées par une double inégalité. L'ensemble Φ'_1 contient alors la contrainte $(\alpha^4 \not\leqslant \alpha_1^5)$. La variable α_1''' étant de la forme β'_m, elle est liée par une double inégalité au type existentiel $\epsilon(\beta'_m, \Phi'_0, [\alpha^l; \alpha^r], \{\beta'_1, \ldots, \beta'_q\})$. La contrainte $(\alpha_1^4 \leqslant \alpha_1''')$ a donc engendré par transitivité $(\alpha_1^4 \leqslant \epsilon(\beta'_m, \Phi'_0, [\alpha^l; \alpha^r], \{\beta'_1, \ldots, \beta'_q\}))$. Comme $\alpha_1^5 = \beta_m''$ alors Φ'_1 contient $(\alpha^4 \not\leqslant \beta_m'')$. Comme précédemment, la saturation de $(\alpha_1^4 \leqslant \epsilon(\beta'_m, \Phi'_0, [\alpha^l; \alpha^r], \{\beta'_1, \ldots, \beta'_q\}))$ par SVarLeqExistOut engendre obligatoirement une contrainte incompatible avec $(\alpha^4 \not\leqslant \beta_m'')$.

⊛ Si α_1^6/α_1^4 sont des β_m''/β'_m et α_1^5/α_1''' sont des α_m''/α'_m.
Similaire au cas précédent.

⊛ Si α_1^6/α_1^4 sont des β_m''/β'_m.
Similaire aux deux cas précédents.

- Les contraintes reliant également β'_i à un type paramétré, c'est-à-dire de la forme $((\alpha_1''', \ldots, \alpha_k''')\ t \leqslant \beta'_i)$, $(\beta'_i \not\leqslant (\alpha_1''', \ldots, \alpha_k''')\ t)$ et $((\alpha_1''', \ldots, \alpha_k''')\ t \not\leqslant \beta'_i)$, sont gérées de manière similaire.
- Si une contrainte est de la forme $(\beta'_i \leqslant \alpha'_j)$, $(\alpha'_j \leqslant \beta'_i)$, $(\beta'_i \not\leqslant \alpha'_j)$ ou $(\alpha'_j \not\leqslant \beta'_i)$, comme la variable universelle β'_i est liée par une double inégalité au type existentiel $\epsilon(\beta'_i, \Phi'_0, [\alpha^l; \alpha^r], \{\beta'_1, \ldots, \beta'_q\})$ et comme α'_i est par construction en dehors de la portée du type existentiel (c'est-à-dire $\alpha'_i \notin [\alpha^l; \alpha^r]$), cette contrainte est gérée exactement comme dans les cas précédents. En effet, la comparaison avec une variable

universelle en dehors de la portée du type existentiel est gérée, par le système de types, de la même manière qu'une comparaison avec un type paramétré.

- Enfin, si une contrainte de $\overline{\Phi'_0}$ est de la forme $(\beta'_i \leqslant \beta'_j)$ ou $(\beta'_i \not\leqslant \beta'_j)$, elle engendre obligatoirement un clash avec les autres contraintes de Φ'_1. Le raisonnement est similaire à celui des cas précédents. On utilise alors les règles SExistLeqExist et SExistNotLeqExist.

Remplacement des sous-arbres de T_6

Il faut également démontrer que le remplacement des sous-arbres de T_6 n'introduit pas de nouvelles contraintes.

Chaque occurrence d'un $\mathtt{x}_i$ dans l'expression $\mathtt{e}$ a engendré dans T_6 un sous-arbre de la forme :

$$\frac{\bigtriangledown}{\Phi'''_1, \Gamma'''_1 \vdash \Psi''' \vee \mathtt{x}_i : \alpha''' \rhd \Phi''''_1}$$

dans laquelle l'environnement de typage Γ'''_1 vérifie $\Gamma'''_1[\mathtt{x}_i] = \beta'_i$.

L'application de la règle d'instanciation a alors généré la contrainte $(\Psi''' \vee \beta'_i \leqslant \alpha''')$. Le sous-arbre T^r_i ayant ajouté la contrainte $(\epsilon(\beta'_i, \Phi'_0, [\alpha^l; \alpha^r], \{\beta'_1, \ldots, \beta'_q\}) \leqslant \beta'_i)$, l'ensemble de contraintes Φ'_1 étant saturé, il contient obligatoirement la contrainte $(\Psi''' \vee \epsilon(\beta'_i, \Phi'_0, [\alpha^l; \alpha^r], \{\beta'_1, \ldots, \beta'_q\}) \leqslant \alpha''')$.

Par ailleurs, après remplacement des $\mathtt{x}_i$ par leur $\mathtt{v}_i$ associé, le sous-arbre correspondant est de la forme :

$$\frac{\bigtriangledown}{\Phi'''_2, \Gamma'''_2 \vdash \Psi''' \vee \mathtt{v} : \alpha''' \rhd \Phi''''_2}$$

dans lequel $\Phi'''_2 \leqslant \Phi'''_1$ et $\Gamma'''_2 \leqslant \Gamma_1$.

Il faut alors démontrer que les contraintes imposées sur α''' dans le nouvel arbre n'engendrent pas plus de contraintes que dans l'ancien. Dans le nouvel arbre, ces contraintes entrent en collision avec celles introduites par le typage de $\mathtt{v}$ tandis que dans l'ancien, elles ont été reportées sur la variable $\epsilon(\beta'_i, \Phi'_0, [\alpha^l; \alpha^r], \{\beta'_1, \ldots, \beta'_q\})$ et gérées par les règles de saturation associées aux types existentiels.

Les contraintes imposées sur α''' dans l'ancien arbre sont soit de la forme $(\alpha''' \leqslant \tau^r)$, soit de la forme $(\tau^r \not\leqslant \alpha''')$. La gestion de ces deux cas est similaire, nous ne nous intéressons ici qu'au premier. Pour qu'un clash provienne dans le nouvel arbre, le type τ^r doit être un type paramétré incompatible (soit par son nom, soit à cause de ses paramètres) avec un type τ^l généré lors du typage de $\mathtt{v} : \alpha'''$ devant vérifier $\tau^l \leqslant \alpha'''$ et donc $\tau^l \leqslant \tau^r$.

Cependant, si τ^r est un type paramétré, puisqu'il doit vérifier la contrainte $\epsilon(\beta'_i, \Phi'_0, [\alpha^l; \alpha^r], \{\beta'_1, \ldots, \beta'_q\}) \leqslant \tau^r$ (à cause de $(\epsilon(\beta'_i, \Phi'_0, [\alpha^l; \alpha^r], \{\beta'_1, \ldots, \beta'_q\}) \leqslant \alpha''')$ et $(\alpha''' \leqslant \tau^r)$), la règle SExistLeqParamed avait obligatoirement été appliquée dans

l'arbre initial, et avait repoussé cette contrainte dans une disjonction reliant τ^r à tous les types de $\not\in\text{RIGHTS}(\beta'_i, \Phi'_0, \{\beta'_1, \ldots, \beta'_q\})$. Néanmoins, chaque type renvoyé par $\not\in\text{RIGHTS}(\beta'_i, \Phi'_0, \{\beta'_1, \ldots, \beta'_q\})$ est soit une variable de type n'appartenant pas à $\{\beta'_1, \ldots, \beta'_q\}$, soit un type paramétré.

S'il s'agit d'une variable de type α'_j, celle-ci est obligatoirement reliée à α''_j par une double inégalité (ajoutée dans les sous-arbres T^r_1, …, T^r_q et T^l_1, …, T^l_q) et Φ''_0 contient obligatoirement la relation $(\beta''_i \leqslant \alpha''_j)$. Comme le typage de $(\mathtt{v} : \beta''_i)$ (via le sous-arbre T^v_i de l'arbre d'origine) avait obligatoirement ajouté la contrainte $(\tau^l \leqslant \beta''_i)$, grâce aux relations $(\beta''_i \leqslant \alpha''_j)$, $(\alpha''_j \leqslant \alpha'_j)$ et $(\alpha'_j \leqslant \tau^r)$, nous en déduisons que l'ensemble de contraintes Φ'_1 contenait obligatoirement déjà $(\tau^l \leqslant \tau^r)$.

S'il s'agit d'un type paramétré $(\ldots)\ \mathtt{t}$ de nom différent de τ^r, la contrainte $((\ldots)\ \mathtt{t} \leqslant \tau^r)$ avait immédiatement engendré un clash dans l'arbre d'origine, et invalidé le membre de la disjonction correspondant.

S'il s'agit d'un type paramétré $(\ldots)\ \mathtt{t}$ de même nom que τ^r, ce type $(\ldots)\ \mathtt{t}$ apparaît obligatoirement aussi (à α-renommage près) dans Φ''_0 dans une relation de la forme $(\beta''_i \leqslant (\ldots)\ \mathtt{t})$. La contrainte $(\tau^l \leqslant (\ldots)\ \mathtt{t})$ était alors également ajoutée dans le sous-arbre T_4 de l'arbre d'origine.

Dans chacun des trois cas précédents, les $\overline{\Psi'}$ introduits par la règle SExistLeqParamed entrent en collision avec les Ψ' correspondants provenant de Φ''_0. Tous les autres membres des disjonctions provenant de Φ''_0 sont alors effectivement supprimés par saturation dans l'arbre d'origine.

Lemme 8 (Les expressions bloquées ne sont pas typables)**.**

Soit $\mathtt{e}_b$ une expression bloquée. Il n'existe aucun schéma de type σ tel que $\mathtt{e}_b : \sigma$.

Démonstration (Lemme 8) :

Comme habituellement, il suffit de lister les différentes formes d'expression bloquées et de démontrer qu'il est impossible d'en dériver un arbre d'inférence. Le nouveau cas intéressant concerne l'utilisation invalide d'un filtrage partiel. Considérons l'expression suivante :

$$\mathtt{match\ K\,(v_1,\ldots,v_n)\ with\ K_1(x_{1,1},\ldots,x_{1,n_1}) \to e_1\ \|\ \ldots\ \|\ K_k(x_{p,1},\ldots,x_{p,n_k}) \to e_k}$$

qui est bloquée lorsque K n'est pas l'un des $\mathtt{K}_i$. Le constructeur K provient donc de la définition d'un type **GADT** de la forme :

$$\mathtt{type}\ (\alpha_1,\ldots,\alpha_p)\ \mathtt{t} = \ldots \mid \exists\beta_1,\ldots,\beta_q\,.\,\mathtt{K}(\beta_1,\ldots,\beta_n)\ [\ \Phi_0\ ]\mid \ldots$$

Lorsque le constructeur K appartient à un autre type **GADT** que celui associé aux $\mathtt{K}_i$, cette expression n'est pas typable pour des raisons évidentes. Par contre, lorsque le constructeur

K appartient au même type **GADT** que les $\mathtt{K}_i$ mais ne correspond à aucun des cas de filtrage, la raison du clash est plus subtile. Dans cette situation, la base de l'arbre d'inférence que l'on tenterait de construire serait obligatoirement de la forme :

$$\dfrac{\overset{T_2}{\triangledown}}{\Phi, \varnothing \vdash \varnothing \vee \alpha_e \leqslant (\alpha'_1, \ldots, \alpha'_p)\ \mathtt{t} \rhd \Phi'}$$

$$\textsc{TMatch}\ \dfrac{\textsc{TConstr}\ \dfrac{\overset{T_1}{\triangledown}}{\varnothing, \varnothing \vdash \varnothing \vee \mathtt{K}\ (\mathtt{v}_1, \ldots, \mathtt{v}_n) : \alpha_e \rhd \Phi} \qquad \textsc{TAbsentCase}\ \dfrac{\overset{T_3}{\triangledown}}{\Phi', \varnothing \vdash \varnothing \vee \mathtt{K} \to \mathtt{CLASH} : \alpha'_1, \ldots, \alpha'_p \rhd \Phi''}}{\varnothing, \varnothing \vdash \varnothing \vee \mathtt{match}\ \mathtt{K}\ (\mathtt{v}_1, \ldots, \mathtt{v}_n)\ \mathtt{with}\ \{\ \mathtt{K}_i(\mathtt{x}_{i,1}, \ldots, \mathtt{x}_{i,n_i}) \to \mathtt{e}_i\ \}_{i=1}^{k} : \alpha \rhd \Phi'}$$

où le sous-arbre T_1 est de la forme :

$$\textsc{TConstr}\ \dfrac{\dfrac{\overset{T_1^v}{\triangledown}}{\Phi_1, \varnothing \vdash \varnothing \vee v_1 : \beta''_1 \rhd \Phi_2} \quad \overset{\ldots}{\ldots} \quad \dfrac{\overset{T_n^v}{\triangledown}}{\Phi_n, \varnothing \vdash \varnothing \vee v_n : \beta''_n \rhd \Phi_{n+1}} \qquad \dfrac{\overset{T_4}{\triangledown}}{\varnothing, \varnothing \vdash \Phi''_0 \rhd \Phi_1} \qquad \dfrac{\overset{T_5}{\triangledown}}{\Phi_{n+1} \vdash (\alpha''_1, \ldots, \alpha''_p)\ \mathtt{t} \leqslant \alpha_e \rhd \Phi}}{\varnothing, \varnothing \vdash \varnothing \vee \mathtt{K}\ (\mathtt{v}_1, \ldots, \mathtt{v}_n) : \alpha_e \rhd \Phi}$$

et le sous-arbre T_3 de la forme :

$$\textsc{TAbsentCase}\ \dfrac{\dfrac{\overset{T_6}{\triangledown}}{\Phi' \vdash \overline{\Phi'_0} \rhd \Phi'_1}}{\Phi', \varnothing \vdash \varnothing \vee \mathtt{K} \to \mathtt{CLASH} : \alpha'_1, \ldots, \alpha'_p \rhd \Phi''}$$

$$\dfrac{\dfrac{\overset{T_1^r}{\triangledown}}{\Phi'_1 \vdash \epsilon(\beta'_1, \Phi'_0, [\alpha^l; \alpha^r], \{\beta'_1, \ldots, \beta'_q\}) \leqslant \beta'_1 \rhd \Phi'_2} \quad \overset{\ldots}{\ldots} \quad \dfrac{\overset{T_q^r}{\triangledown}}{\Phi'_q \vdash \epsilon(\beta'_q, \Phi'_0, [\alpha^l; \alpha^r], \{\beta'_1, \ldots, \beta'_q\}) \leqslant \beta'_q \rhd \Phi'_{q+1}}}{\dfrac{\overset{T_1^l}{\triangledown}}{\Phi'_{q+1} \vdash \beta'_1 \leqslant \epsilon(\beta'_1, \Phi'_0, [\alpha^l; \alpha^r], \{\beta'_1, \ldots, \beta'_q\}) \rhd \Phi'_{q+2}} \quad \overset{\ldots}{\ldots} \quad \dfrac{\overset{T_q^l}{\triangledown}}{\Phi'_{2q} \vdash \beta'_q \leqslant \epsilon(\beta'_q, \Phi'_0, [\alpha^l; \alpha^r], \{\beta'_1, \ldots, \beta'_q\}) \rhd \Phi''}}$$

avec $\Phi'_0 = \Phi_0[\alpha_i \mapsto \alpha'_i]_{i=1}^{p}[\beta_i \mapsto \beta'_i]_{i=1}^{q}$ et $\Phi''_0 = \Phi_0[\alpha_i \mapsto \alpha''_i]_{i=1}^{p}[\beta_i \mapsto \beta''_i]_{i=1}^{q}$.

Nous remarquerons que le sous-arbre T_4 tente d'ajouter Φ''_0 dans l'ensemble de contraintes et que T_6 tente d'y ajouter $\overline{\Phi'_0}$. Or Φ'_0 et Φ''_0 sont tous les deux obtenus à partir de Φ_0 par renommage des variables $\{ \alpha_i \}_{i=1}^{p}$ et $\{ \beta_i \}_{i=1}^{q}$.
Nous nous retrouvons alors exactement dans la même situation que dans la preuve du lemme 5 lorsque le typage du corps d'un cas de filtrage provoquait un clash dans lequel nous avions montré que l'ajout de Φ''_0 et $\overline{\Phi'_0}$ dans le même ensemble de contraintes provoquait obligatoirement un clash.

Grâce à ces lemmes, la démonstration des théorèmes de validité est la même que pour le système de base.

5.6 Gestion de la récursion polymorphe

5.6.1 Motivation

La définition d'une fonction récursive polymorphe, que ce soit en présence de **GADT** ou non, est en général refusée par les systèmes que nous avons définis jusqu'à maintenant. En effet, l'utilisation de nos règles d'inférence sur un combinateur de point fixe ne permet pas, dans le cadre général, d'accepter une fonction récursive s'utilisant elle-même plusieurs fois dans son corps dans des « contextes de typage » différents.

Pour accepter la définition de telles fonctions, la technique classique, que nous reprenons ici, consiste à demander au programmeur d'annoter sa fonction récursive polymorphe avec un schéma de type. Nous étendons alors le langage avec une construction « `let rec` » comme suit :

`e +:= let rec (f :` σ`) x =` e_1 `in` e_2

où « `f` » et « `x` » sont des variables, et « σ » est le schéma de type donné par le programmeur pour la fonction `f`. La variable `f` peut alors apparaître libre dans e_1. Une telle définition est sémantiquement équivalente à :

`let f = fix (`λ `f x .` e_1`) in` e_2

où « `fix` » est un combinateur de point fixe.

Lors du typage du corps (e_1) d'une telle fonction `f` définie via un `let rec`, pour chaque usage de `f` dans e_1, le typeur pourra instancier, avec des variables de type fraîches, le schéma de type σ donné par le programmeur. Ce n'est néanmoins pas suffisant pour assurer la correction du programme, il faut également vérifier que l'annotation de type qu'a donnée le programmeur est « valide » vis-à-vis du code de `f`. Pour ce faire, il suffit au typeur de généraliser le type qu'il a calculé pour `f` et de vérifier que le schéma de type qu'avait donné le programmeur est « moins général » (plus précis) que celui qu'il obtient par cette généralisation.

Cependant, répondre à la question « un schéma de type est-il moins général qu'un autre ? » n'est pas trivial, en particulier dans notre contexte dans lequel les schémas peuvent contenir des contraintes arbitraires de sous-typage et de non-sous-typage liées par des opérateurs logiques de disjonction et de conjonction.

5.6.2 Idée

Intuitivement, pour vérifier que la définition d'une fonction récursive polymorphe `f` est valide, il suffit de vérifier que toutes les contraintes engendrées par le typage du corps de `f` peuvent être déduites de celles présentes dans le schéma de type qu'a donné le programmeur pour `f`. Cette situation est très similaire au typage d'un cas de filtrage sur un **`GADT`** lors duquel il faut vérifier que les contraintes imposées sur les types existentiels lors du typage du corps du cas peuvent être déduites de l'ensemble de contraintes donné par le programmeur lors de la définition du constructeur de **`GADT`** correspondant à ce cas.

Pour vérifier qu'un schéma de type $[\ \forall\alpha_1 \ldots \alpha_n \,.\, \alpha_0 \mid \Phi_0\ ]$ est moins général qu'un autre schéma de type $[\ \forall\alpha'_1 \ldots \alpha'_p \,.\, \alpha'_0 \mid \Phi'_0\ ]$, nous allons alors :

1. Générer n types existentiels associés aux n variables $\alpha_1, \ldots, \alpha_n$ et utilisant Φ_0 comme ensemble de contraintes.
2. Instancier le second schéma ($[\ \forall\alpha'_1 \ldots \alpha'_p \,.\, \alpha'_0 \mid \Phi'_0\ ]$).
3. Imposer une double inégalité entre le type existentiel associé à α_0 et la variable universelle obtenue par instanciation de α'_0.
4. Tenter de saturer l'ensemble de contraintes obtenu contenant en particulier cette double inégalité et les contraintes provenant de l'instanciation de Φ'_0.

Si la saturation fonctionne, nous aurons alors prouvé que le schéma $[\ \forall\alpha_1 \ldots \alpha_n \,.\, \alpha_0 \mid \Phi_0\ ]$ est moins général que $[\ \forall\alpha'_1 \ldots \alpha'_p \,.\, \alpha'_0 \mid \Phi'_0\ ]$, ce que nous notons :

$$[\ \forall\alpha'_1 \ldots \alpha'_p \,.\, \alpha'_0 \mid \Phi'_0\ ] \preccurlyeq [\ \forall\alpha_1 \ldots \alpha_n \,.\, \alpha_0 \mid \Phi_0\ ]$$

Nous étendons alors le système de types par la règle de typage suivante :

$$\textsc{TLetRec}\quad \frac{\texttt{let}\ \alpha'\ \textit{fresh} \qquad \Phi, \Gamma \oplus (\texttt{f} : \sigma) \vdash \Psi \vee \lambda\ \texttt{x}\ .\ \texttt{e}_1 : \alpha' \rhd \Phi' \qquad \Phi' \vdash \Psi \vee \textsc{gen}(\alpha', \Phi', \Gamma \oplus (\texttt{f} : \sigma)) \preccurlyeq \sigma \rhd \Phi'' \qquad \Phi'', \Gamma \oplus \Psi \vee (\texttt{f} : \sigma) \vdash \texttt{e}_2 : \alpha \rhd \Phi'''}{\Phi, \Gamma \vdash \Psi \vee \texttt{let rec}\ (\texttt{f} : \sigma)\ \texttt{x} = \texttt{e}_1\ \texttt{in}\ \texttt{e}_2 : \alpha \rhd \Phi'''}$$

et la règle suivante gérant la comparaison entre schémas :

$$\textsc{SCmpSchema}\quad \frac{\begin{array}{c}\texttt{let } \alpha^l \textit{ fresh} \qquad \texttt{let } \alpha''_1,\ldots,\alpha''_p,\beta_1,\ldots,\beta_n \textit{ fresh} \qquad \texttt{let } \alpha^r \textit{ fresh} \\ \texttt{let } \Phi'_1 = \Phi'_0[\alpha'_i \mapsto \alpha''_i]_{i=1}^{p} \qquad \texttt{let } \Phi_1 = \Phi_0[\alpha_i \mapsto \beta_i]_{i=1}^{n} \\ \Phi \vdash \Psi \vee \Phi'_1 \rhd \Phi 1 \\ \Phi' \vdash \Psi \vee \alpha'_0[\alpha'_i \mapsto \alpha''_i]_{i=1}^{p} \leqslant \alpha_0[\alpha_i \mapsto \beta_i]_{i=1}^{n} \rhd \Phi'' \\ \Phi'' \vdash \Psi \vee \alpha_0[\alpha_i \mapsto \beta_i]_{i=1}^{n} \leqslant \alpha'_0[\alpha'_i \mapsto \alpha''_i]_{i=1}^{p} \rhd \Phi''' \\ \Phi''' \vdash \{\ \Psi \vee \epsilon(\beta_i, \Phi_1, [\alpha^l; \alpha^r], \{\beta_1,\ldots,\beta_n\}) \leqslant \beta_i\ \}_{i=1}^{n} \rhd \Phi'''' \\ \Phi'''' \vdash \{\ \Psi \vee \beta_i \leqslant \epsilon(\beta_i, \Phi_1, [\alpha^l; \alpha^r], \{\beta_1,\ldots,\beta_n\})\ \}_{i=1}^{n} \rhd \Phi''''' \end{array}}{\Phi \vdash \Psi \vee [\ \forall \alpha'_1 \ldots \alpha'_p\ .\ \alpha'_0 \mid \Phi'_0\ ] \preccurlyeq [\ \forall \alpha_1 \ldots \alpha_n\ .\ \alpha_0 \mid \Phi_0\ ] \rhd \Phi'''''}$$

dans laquelle la première prémisse ($\Phi_1 \vdash \Psi \vee \Phi'_1 \rhd \Phi_2$) correspond à l'instanciation du schéma $[\ \forall \alpha'_1 \ldots \alpha'_p\ .\ \alpha'_0 \mid \Phi'_0\ ]$, les deux prémisses suivantes imposent la double inégalité entre l'instanciation de α'_0 et la variable universelle associée à α_0, et les deux dernières imposent la double inégalité entre les variables universelles associées à $\alpha_0, \ldots, \alpha_n$ et les types existentiels correspondants.

5.6.3 Exemple simple

Commençons par observer le comportement de cette technique sur un exemple simple. Pour ce faire, considérons la définition suivante d'une fonction récursive polymorphe nommée `id` :

```
let rec (id : [ ∀α α' . α | α' → α' ⩽ α ]) x =
  if rand () then (
    print_int (id 3);
    print_string (id "trois");
  );
  x in
...
```

dans laquelle :

- La fonction « `rand` » renvoie simplement un booléen de manière aléatoire. Nous l'utilisons ici uniquement pour faire potentiellement terminer l'exécution de la fonction sans ajouter de complexité inutile au code.
- Les fonctions « `print_int` » et « `print_string` » affichent respectivement un entier et une chaîne de caractères.
- La construction « `if` e_1 `then` e_2 » est du sucre syntaxique sur « `if` e_1 `then` e_2 `else ()` ».
- La construction de séquence « e_1`;` e_2 » est du sucre syntaxique sur « `let x =` e_1 `in` e_2 » où « `x` » est une variable inutilisée ailleurs dans le programme.

Les deux appels à la fonction `id` se font avec des arguments de types différents (un entier et une chaîne). Leurs typages fonctionnent néanmoins puisque chaque utilisation de `id` provoque une instanciation du schéma de type :

$$\sigma = [\ \forall \alpha\, \alpha' \,.\, \alpha \mid \alpha' \to \alpha' \leqslant \alpha\ ]$$

et donc la génération de variables de type distinctes.

Par ailleurs, à la fin du typage de la fonction `id`, le schéma de type inféré est :

$$\sigma' = [\ \forall \alpha_0\, \alpha_1\, \alpha_2 \,.\, \alpha_0 \mid \alpha_1 \to \alpha_2 \leqslant \alpha_0 \ \wedge\ \alpha_2 \leqslant \alpha_1\ ]$$

Nous y avons omis les éventuelles contraintes parasites provenant du typage des appels à `rand`, `print_int` et `print_string` qui, grâce à un algorithme intelligent de nettoyage des schémas, disparaîtraient de toute manière.

Pour vérifier que σ' est moins général que σ, nous générons alors deux variables universelles β et β' liées à deux types existentiels de la manière suivante :

- $\beta \leqslant \epsilon(\beta,\ (\beta' \to \beta' \leqslant \beta), [\alpha^l; \alpha^r], \{\beta, \beta'\}) \leqslant \beta$
- $\beta' \leqslant \epsilon(\beta', (\beta' \to \beta' \leqslant \beta), [\alpha^l; \alpha^r], \{\beta, \beta'\}) \leqslant \beta'$

L'instanciation de σ', en suivant le renommage $[\alpha_0 \mapsto \alpha'_0, \alpha_1 \mapsto \alpha'_1, \alpha_2 \mapsto \alpha'_2\ ]$, engendre les contraintes :

- $\alpha'_1 \to \alpha'_2 \leqslant \alpha'_0$
- $\alpha'_2 \leqslant \alpha'_1$

Nous imposons alors la double inégalité reliant le schéma inféré au schéma imposé par l'utilisateur :

- $\alpha'_0 \leqslant \beta \leqslant \alpha'_0$

À cause de la transitivité de $(\leqslant)$, nous obtenons :

- $\alpha'_1 \to \alpha'_2 \leqslant \epsilon(\beta,\ (\beta' \to \beta' \leqslant \beta), [\alpha^l; \alpha^r], \{\beta, \beta'\})$

En utilisant la règle de saturation SExistLeqParamed, cette comparaison entre un type paramétré et un type existentiel engendre la contrainte :

- $\beta' \to \beta' \leqslant \alpha'_1 \to \alpha'_2$

La règle SLeqSameParamed engendre alors les deux doubles-inégalités :

- $\beta' \leqslant \alpha'_1 \leqslant \beta'$
- $\beta' \leqslant \alpha'_2 \leqslant \beta'$

À cause de la transitivité de $(\leqslant)$, nous obtenons une contrainte de sous-typage entre deux fois le même type existentiel :

- $\epsilon(\beta', (\beta' \to \beta' \leqslant \beta), [\alpha^l; \alpha^r], \{\beta, \beta'\}) \leqslant \epsilon(\beta', (\beta' \to \beta' \leqslant \beta), [\alpha^l; \alpha^r], \{\beta, \beta'\})$

acceptée par l'axiome SLeqSameExist.

La saturation se termine alors sans erreur.

En revanche, si le schéma de type donné par le programmeur était :

$$\sigma = [\ \forall \alpha\, \alpha', \alpha''\ .\ \alpha \mid \alpha' \to \alpha'' \leqslant \alpha\ ]$$

la dernière contrainte obtenue aurait simplement été entre deux types existentiels distincts, et l'ensemble de contraintes qu'elles contiendraient ne les relieraient pas. L'application de la règle SExistLeqExist engendrerait alors une disjonction vide, et donc un clash de typage.

5.6.4 Exemple avancé

Un exemple intéressant mettant en évidence tout l'intérêt des **GADT** et de la récursion polymorphe est celui bien connu de la fonction d'évaluation bien typée. Il consiste à définir un type **GADT** représentant un arbre de syntaxe :

```
type α expr =
| Int int    [ int  ⩽ α ]
| Bool bool  [ bool ⩽ α ]
| Ite(β1,β2,β3) [ β1 = β4 expr ∧ β4 = bool ∧ β2 = α expr ∧ β3 = α expr ]
```

dans lequel les $(=)$ sont du sucre syntaxique sur de doubles inégalités.

Le paramètre du type `expr` représente le type de l'expression associée à l'arbre de syntaxe. Le typage des **GADT** nous interdit alors de construire une expression invalide du point de vue du typage du langage encodé par l'arbre de syntaxe comme par exemple :

```
Ite (Int 42, Bool true, Bool false)
```

et, dans le cas d'une expression valide, comme :

```
Ite (Bool true, Int 3, Ite (Bool false, Int 4, Int 5))
```

le schéma de type inféré automatiquement est :

$$[\ \forall\, \alpha_0\, \alpha_1\ .\ \alpha_0 \mid \texttt{int} \leqslant \alpha_1 \ \wedge\ \alpha_1\ \texttt{expr} \leqslant \alpha_0\ ]$$

dans lequel le `int` correspond effectivement au type de l'expression représentée par l'arbre de syntaxe.

La fonction d'évaluation pour un tel arbre de syntaxe peut alors s'écrire de la manière suivante :

```
let rec (eval : [
  ∀ α_arg α_res α_annot α . α |
    α_arg → α_res ⩽ α  ∧  α_arg ⩽ α_annot expr ∧
    int ⩽ α_annot ⇒ int ⩽ α_res  ∧  bool ⩽ α_annot ⇒ bool ⩽ α_res
]) e =
  match e with
  | Int n -> n
  | Bool b -> b
  | Ite (test, ifso, ifnot) -> if eval test then eval ifso
                               else eval ifnot
```

où l'opérateur logique ($\Rightarrow$) représente une implication. Il s'agit simplement de sucre syntaxique : la propriété ($\texttt{int} \leqslant \alpha_{annot} \Rightarrow \texttt{int} \leqslant \alpha_{res}$) est expansée en ($\texttt{int} \not\leqslant \alpha_{annot} \vee \texttt{int} \leqslant \alpha_{res}$).

Le typage de cette définition de la fonction `eval` est similaire à celui de l'exemple précédent, avec juste un peu plus de variables de type à manipuler. Le lecteur pourra se servir de l'implémentation disponible en ligne et décrite dans le prochain chapitre pour en vérifier le fonctionnement en pratique.

Chapitre 6

Implémentation

Le formalisme que nous avons utilisé pour définir les différents systèmes de types de cette thèse permet de les implémenter directement. Pour typer une expression e, il suffit en effet de tenter de construire l'arbre d'inférence grâce aux règles fournies par le système considéré. À chaque étape de la construction de cet arbre, ou bien aucune règle ne s'applique et le typage s'arrête sur une erreur, ou alors une unique règle s'applique, et le typage continue de manière déterministe. Cette propriété est vraie pour la « partie typage » de l'arbre comme pour la « partie saturation ». L'arbre à lui seul représente la preuve complète de correction de l'expression. De plus, chaque règle mentionne précisément les variables de type universelles qu'il faut générer lors de l'application de chacune des règles et il n'y a jamais d'opération implicite à effectuer comme par exemple un renommage global des variables de type dans l'arbre d'inférence.

Cette propriété du formalisme utilisé présente beaucoup d'avantages pratiques. Tout d'abord, elle permet de se convaincre rapidement qu'un ensemble de règles « a du sens » (et n'est pas ambigu, en particulier), qu'il s'agisse des règles de typage ou des règles de saturation ; mais surtout, son implémentation dans un but de prototypage du système est triviale puisque complètement automatique. Il serait d'ailleurs intéressant d'implémenter un « compilateur » prenant en entrée un ensemble de règles d'inférence dans ce formalisme et générant un programme effectuant la synthèse de types correspondante. Cela permettrait de rendre encore plus simple le prototypage de tels systèmes de types en rendant leur description « exécutable ».

Pour des systèmes classiques à la ML, tout comme pour notre système de base présenté dans le chapitre 2, une telle technique d'implémentation naïve fonctionne très bien et passe raisonnablement à l'échelle sur de vrais programmes. Le typeur obtenu est bien entendu moins performant que ceux, adaptés aux systèmes à base d'unification, utilisant des structures mutables pour représenter les variables de type et le fameux algorithme « union-find » (cf. [CP15, H91]). Nous remarquerons néanmoins qu'un tel algorithme est spécifique à l'unification et ne peut pas être utilisé directement pour gérer le sous-typage.

Malheureusement, pour des systèmes plus complexes comme ceux présentés dans les chapitres 3, 4 et 5, cette technique naïve d'implémentation pose d'énormes problèmes de performance. Ceci est principalement dû à trois raisons :

1. La présence de disjonctions, dans les systèmes des chapitres 3 et 5, pose des problèmes d'explosion combinatoire dans les mécanismes de saturation à cause des règles dites de « backtrack ».

2. Le mécanisme de généralisation qui, en encapsulant systématiquement l'ensemble des contraintes, provoque une explosion du nombre de variables de type générées à chaque instanciation ainsi qu'une explosion du nombre de contraintes manipulées tout au long du typage.
3. Le mécanisme de saturation qui, en laissant dans Φ des contraintes impliquées par d'autres (comme par exemple $(\alpha \leqslant \texttt{int} \vee \beta \leqslant \texttt{string})$ qui est impliquée par $(\beta \leqslant \texttt{string})$), engendre une explosion du nombre d'éléments renvoyés par les fonctions LEFTS, RIGHTS, etc. et ainsi une explosion du temps de calcul et du nombre de contraintes inutiles manipulées.

Un autre inconvénient de cette technique naïve d'implémentation concerne la lisibilité des schémas de type restitués à l'utilisateur. En effet, la taille des schémas a tendance à exploser rapidement pour les mêmes raisons que celles présentées ci-dessus, et un simple nettoyage par analyse de dépendances ne suffit pas en pratique pour rendre leur lecture compréhensible par un humain.

Malheureusement, la « partie typage » de l'algorithme peut difficilement être améliorée puisqu'elle consiste en un simple parcours récursif du programme. De plus, le typage et la saturation, comme nous les avons définis dans nos différents systèmes, extraient en général le minimum de contraintes permettant d'assurer la validité du code. Il est bien entendu possible d'éviter la génération d'une ou deux variables de type par endroit, voire de contracter certaines contraintes en une seule dans quelques cas particuliers, mais cela n'a presque aucun impact sur les performances globales. Enfin, utiliser une représentation arborescente des types (en autorisant des constructions comme par exemple $(\alpha_1 \to \alpha_2 \to \alpha_3)$), plutôt que plate comme ici, peut permettre d'éviter quelques indirections dans leur représentation en mémoire, mais n'apporte algorithmiquement rien puisque les tests à effectuer sont au final toujours les mêmes. Une telle représentation risque même de faire perdre en performance puisqu'elle diminue les possibilités de « mémorisation » intrinsèques au calcul de point fixe effectué lors de la saturation.

Les deux seuls moyens que nous ayons trouvés pour améliorer significativement les performances de nos implémentations sont :

- Nettoyer les ensembles de contraintes contenus dans les schémas de type. Pour ce faire, nous présentons trois techniques orthogonales dans les sections 6.1.2, 6.1.3 et 6.1.4.
- Utiliser une représentation dédiée des ensemble de contraintes contenant des disjonctions. Celle-ci permet d'optimiser les primitives utilisées par les algorithmes de nettoyage et de limiter certains calculs redondants lors de la saturation. Elle est décrite dans la section 6.2.

Ces techniques permettent alors de réduire suffisamment les problèmes de performance pour rendre nos systèmes de types utilisables sur de vrais programmes, y compris le système du chapitre 5 qui a été le plus compliqué à implémenter. Elles ont été testées et semblent bien fonctionner en pratique. Elles peuvent faire gagner jusqu'à plusieurs ordres de grandeur sur le temps de calcul et la taille des ensembles de contraintes. Les mécanismes de nettoyage ont en plus l'avantage de rendre les schémas de type plus lisibles pour le programmeur moyen.

6.1 Nettoyage des ensembles de contraintes

6.1.1 Principe de base

De manière générale, les techniques de nettoyage que nous présentons ici consistent en la suppressions de certaines contraintes dont on sait qu'elles seront « inutiles », non pas au sens où, si on les conservait, elles ne seraient plus utilisées lors de la saturation ; mais au sens où leur suppression ne modifie pas le fait qu'un programme est accepté ou refusé par le typeur.

Contrairement à ce que la notion de « nettoyage » suggère, le but premier n'est pas ici de diminuer l'occupation mémoire, mais de faire chuter le temps de calcul. En effet, en réduisant la taille des ensembles de contraintes, on diminue en particulier le nombre d'éléments renvoyés par les fonctions LEFTS, RIGHTS, etc. et ainsi le nombre d'opérations à effectuer lors de la saturation. Par ailleurs, diminuer la taille des ensembles de contraintes compris dans les schémas de type permet de diminuer le nombre de contraintes réintroduites à chaque instanciation et le nombre de variables de type fraîches générées. Il s'agit donc d'une opération nécessaire pour éviter une explosion combinatoire du nombre de contraintes.

Il ne faut bien entendu pas nettoyer les ensembles de contraintes à n'importe quel moment du typage ou de la saturation. En effet, dans certaines règles, comme par exemple :

$$\textsc{TApp}\quad \frac{\texttt{let}\ \alpha_1, \alpha_2, \alpha_3\ \textit{fresh} \qquad \Phi \vdash \alpha_1 \leqslant \alpha_2 \to \alpha_3 \rhd \Phi' \qquad \Phi' \vdash \alpha_3 \leqslant \alpha \rhd \Phi'' \qquad \Phi'', \Gamma \vdash \mathtt{e}_1 : \alpha_1 \rhd \Phi''' \qquad \Phi''', \Gamma \vdash \mathtt{e}_2 : \alpha_2 \rhd \Phi''''}{\Phi, \Gamma \vdash \mathtt{e}_1\ \mathtt{e}_2 : \alpha \rhd \Phi''''}$$

certains ensembles de contraintes intermédiaires (comme le Φ' de la règle ci-dessus) contiennent des contraintes qui ne sont pas encore liées aux autres (comme $(\alpha_1 \leqslant \alpha_2 \to \alpha_3)$). Une simple analyse de dépendances pourrait les considérer « inatteignables » et les supprimer par erreur.

Le nettoyage des contraintes peut par contre être effectué sans risque lors d'une généralisation, typiquement lors du typage d'un `let`. Pour ce faire, il suffit d'implémenter les différents algorithmes que nous allons décrire maintenant dans la fonction de généralisation (GEN) afin de nettoyer le schéma de type qu'elle génère.

6.1.2 Nettoyage par analyse de dépendances

Le nettoyage par analyse de dépendances consiste simplement à ne conserver de l'ensemble de contraintes Φ_0 contenu dans un schéma de type $[\,\forall\alpha_1 \ldots \alpha_n \,.\, \alpha_0 \mid \Phi_0\,]$ que les contraintes susceptibles d'être utilisées par l'algorithme de saturation après une instanciation de ce schéma.

Le point d'entrée du schéma est la variable α_0. C'est sur elle (ou plus précisément sur son image par le renommage des variables effectué par l'instanciation) que l'algorithme de saturation est susceptible d'imposer une contrainte pouvant se propager aux autres contraintes de Φ_0 liées à α_0. L'analyse de dépendances cherchera donc à extraire de Φ_0 les contraintes atteignables par saturation depuis α_0.

Analyse de dépendances naïve

L'ensemble de contraintes Φ_0 est, par définition, sous forme normale conjonctive. À la base, on cherche à n'en conserver que certaines disjonctions. En effet, la gestion des disjonctions dans l'algorithme de saturation fait que, si un membre d'une disjonction est invalidée, la saturation est rabattue sur les autres membres. À partir du moment où un membre d'une disjonction est « atteignable », les autres le sont donc aussi.

Ces disjonctions sont reliées entres elles par des variables de type. On peut les représenter sous forme d'un graphe dont les noeuds sont les disjonctions composant Φ_0 et chaque arête entre deux noeuds correspond à l'existence d'une variable de type commune aux deux disjonctions en question. L'ensemble des disjonctions à conserver de Φ_0 est donc naïvement composé des différents noeuds contenus dans les composantes connexes du graphe mentionnant α_0.

Cette technique permet de supprimer les disjonctions qui sont présentes dans Φ_0 mais qui n'ont aucun lien avec α_0. De telles disjonctions apparaissent typiquement lors du typage d'un code comme :

```
let x =
  print_string "Initialisation de x";
  42 in
...
```

L'ensemble des contraintes générées lors du typage de la définition de `x` contient alors :

- $\alpha_1 \to \alpha_2 \leqslant \alpha_3$
- $\alpha_1 \leqslant$ `string`
- `unit` $\leqslant \alpha_2$

} générées lors du typage de la primitive `print_string`

- `string` $\leqslant \alpha_4$

} générée lors du typage de la constante `"Initialisation de x"`

- $\alpha_3 \leqslant \alpha_5 \to \alpha_6$
- $\alpha_6 \leqslant \alpha_7$
- $\alpha_4 \leqslant \alpha_5$

} générées lors du typage de l'appel de primitive

- `int` $\leqslant \alpha_8$
- $\alpha_8 \leqslant \alpha_0$

} générées lors du typage de la constante `42`

- $\alpha_1 \to \alpha_2 \leqslant \alpha_5 \to \alpha_6$
- $\alpha_1 \leqslant \alpha_5$
- $\alpha_5 \leqslant \alpha_1$
- $\alpha_6 \leqslant \alpha_2$
- $\alpha_2 \leqslant \alpha_6$
- `string` $\leqslant \alpha_5$
- `string` $\leqslant \alpha_1$
- `string` $\leqslant$ `string`
- `unit` $\leqslant \alpha_6$
- `unit` $\leqslant \alpha_7$
- `int` $\leqslant \alpha_0$

} générées par saturation

Les contraintes générées lors du typage de l'appel à `print_string`, servant en particulier à vérifier que le type de la chaîne passée en argument est compatible avec le type du paramètre de `print_string`, peuvent alors être supprimées lors de la génération du schéma de type associé à la variable `x`. Le schéma de type peut ainsi être contracté en :

$$[\ \forall\, \alpha_0\, \alpha_8\, .\, \alpha_0 \mid \texttt{int} \leqslant \alpha_0 \ \wedge\ \texttt{int} \leqslant \alpha_8 \ \wedge\ \alpha_8 \leqslant \alpha_0\ ]$$

Cette technique est bien entendu très naïve et peut être améliorée sur de nombreux points.

Suppression des variables intermédiaires

Dans un schéma de type de la forme $[\ \forall \alpha_1 \ldots \alpha_n\ .\ \alpha_0 \mid \Phi_0\ ]$, une variable de type α est dite « intermédiaire » si elle est généralisée (c'est-à-dire si $\alpha \in \{\alpha_1, \ldots, \alpha_n\}$), différente de α_0 et si elle apparaît dans Φ_0 uniquement seule à gauche ou à droite d'une relation de sous-typage ou de non-sous-typage (et donc en particulier jamais comme paramètre d'un type paramétré).

Toutes les disjonctions ne contenant que des contraintes de la forme $\alpha \leqslant \tau^r$, $\tau^l \leqslant \alpha$, $\alpha \not\leqslant \tau^l$ ou $\tau^r \not\leqslant \alpha$ avec α une variable intermédiaire peuvent alors être supprimées, ainsi que les variables intermédiaires elles-mêmes.

Cette amélioration permet en particulier de réduire le schéma de type de l'exemple précédent à :

$$[\ \forall\, \alpha_0\, .\, \alpha_0 \mid \texttt{int} \leqslant \alpha_0\ ]$$

en supprimant la variable intermédiaire α_8 et les contraintes $(\texttt{int} \leqslant \alpha_8)$ et $(\alpha_8 \leqslant \alpha_0)$ associées.

Bien entendu, un tel nettoyage n'est possible que parce que l'ensemble de contraintes Φ_0 contenu dans le schéma de type est saturé.

Utilisation de la position des variables dans les contraintes

La relation de dépendance que nous avons définie précédemment entre deux disjonctions peut être affinée en prenant en compte la position des variables dans les types contenus dans les contraintes.

Nous pouvons en effet remarquer que les algorithmes de saturation correspondant aux différents systèmes que nous avons définis dans cette thèse n'accèdent à l'ensemble de contraintes Φ en cours de saturation que par l'usage des fonctions LEFTS, RIGHTS, etc. Cependant, ces fonctions prennent une variable de type α en argument et n'extraient de Φ que les contraintes mentionnant directement α seul à gauche ou à droite d'une relation de sous-typage ou de non-sous-typage. Ainsi, les contraintes comme par exemples $(\alpha' \leqslant \alpha \to \alpha)$, ne mentionnant α que comme paramètre d'un constructeur de type, ne sont jamais renvoyées par un appel à LEFTS, RIGHTS, etc. avec α en argument.

Avant d'affiner la relation de dépendance entre deux disjonctions de contraintes, quelques définitions sont nécessaires.

1. Nous dirons qu'une contrainte $\mathtt{C}$ **dépend directement** d'une variable de type α si $\mathtt{C}$ est de l'une des cinq formes suivantes :
   - $\alpha \leqslant \tau^r$
   - $\tau^l \leqslant \alpha$
   - $\alpha \not\leqslant \tau^l$
   - $\tau^r \not\leqslant \alpha$
   - $\alpha \# \mathtt{K}$
2. Nous dirons qu'une contrainte $\mathtt{C}$ **dépend de l'extérieur** d'un schéma de type de la forme $[\ \forall\alpha_1 \ldots \alpha_n\ .\ \alpha_0 \mid \Phi_0\ ]$ si elle dépend directement d'une variable de type α qui n'est pas généralisée (c'est-à-dire vérifiant $(\alpha \notin \{\ \alpha_1, \ldots, \alpha_n\ \})$).
3. Nous dirons qu'une contrainte $\mathtt{C}$ **dépend indirectement** d'une variable de type α dans un schéma de type σ si α est libre dans $\mathtt{C}$ et si $\mathtt{C}$ dépend de l'extérieur de σ.
4. Nous dirons qu'une contrainte $\mathtt{C}$ **dépend** d'une variable de type α dans un schéma de type σ si elle en dépend directement ou indirectement dans σ.
5. Nous dirons qu'une disjonction de contraintes $(\mathtt{C}_1 \vee \cdots \vee \mathtt{C}_p)$ **dépend** d'une variable de type α dans un schéma de type σ si au moins l'un des $\mathtt{C}_i$ dépend de α dans σ.
6. Nous dirons qu'une disjonction de contraintes $(\mathtt{C}_1 \vee \cdots \vee \mathtt{C}_p)$ **dépend** d'une autre disjonction de contrainte $(\mathtt{C}'_1 \vee \cdots \vee \mathtt{C}'_q)$ dans un schéma de type σ s'il existe une variable de type α libre dans $(\mathtt{C}'_1 \vee \cdots \vee \mathtt{C}'_q)$ telle que $(\mathtt{C}_1 \vee \cdots \vee \mathtt{C}_p)$ dépend de α dans σ.

Cette dernière définition de « dépendance » entre disjonctions de contraintes est asymétrique (au sens ou une disjonction de contraintes ψ_1 peut dépendre d'une autre disjonction ψ_2 sans que ψ_2 dépende de ψ_1) contrairement à la précédente qui était symétrique.

Pour nettoyer un schéma de type $[\ \forall\alpha_1 \ldots \alpha_n\ .\ \alpha_0 \mid \Phi_0\ ]$, il suffit alors de ne conserver de Φ_0 que les disjonctions qui dépendent de α_0, les disjonctions dépendant de ces disjonctions, et ainsi de suite

Grâce à cette optimisation, le schéma de type :

$$[\ \forall\alpha_0\alpha_1\alpha_2\ .\ \alpha_0 \mid \mathtt{int} \leqslant \alpha_0\ \wedge\ \alpha_0 \to \alpha_1 \leqslant \alpha_2\ ]$$

peut être simplifié en :

$$[\ \forall\alpha_0\ .\ \alpha_0 \mid \mathtt{int} \leqslant \alpha_0\ ]$$

En revanche, le schéma de type :

$$[\ \forall\alpha_0\alpha_1\ .\ \alpha_0 \mid \mathtt{int} \leqslant \alpha_0\ \wedge\ \alpha_0 \to \alpha_1 \leqslant \alpha_2\ ]$$

où α_2 n'est pas généralisé, ne peut pas être simplifié.

Ceci est parfaitement normal puisqu'une contrainte apparaissant ultérieurement sur α_2 pourrait contraindre α_0 via la relation $(\alpha_0 \to \alpha_1 \leqslant \alpha_2)$. Cette dernière ne peut donc pas être supprimée sans mettre en péril la validité du typage.

6.1.3 Nettoyage grâce à la rotation des disjonctions

Définitions préliminaires

Dans cette section, nous dirons qu'« une contrainte C_1 entre en collision avec une autre contrainte C_2 » lorsqu'il existe une règle de saturation qui, appliquée sur C_1 avec C_2 dans Φ (ou sur C_2 avec C_1 dans Φ, ce qui est équivalent puisque les règles de saturation gèrent systématiquement les deux situations pour rester déterministe par rapport à l'ordre d'arrivée des contraintes) engendre une nouvelle contrainte C_3.

Par exemple, $(\texttt{int} \leqslant \alpha)$ et $(\alpha \leqslant \alpha')$ entrent en collision et génèrent $(\texttt{int} \leqslant \alpha')$. De même, $(\texttt{int} \not\leqslant \alpha)$ et $(\texttt{string} \leqslant \alpha)$ entrent en collision et génèrent $(\texttt{int} \not\leqslant \texttt{string})$.

De manière similaire, lorsqu'un membre C_i d'une disjonction $(\mathsf{C}_1 \vee \cdots \vee \mathsf{C}_n)$ entre en collision avec le membre C'_j d'une disjonction $(\mathsf{C}'_1 \vee \cdots \vee \mathsf{C}'_p)$, la saturation engendre une disjonction de contraintes contenant :

- tous les membres de $(\mathsf{C}_1 \vee \cdots \vee \mathsf{C}_n)$ sauf C_i
- tous les membres de $(\mathsf{C}'_1 \vee \cdots \vee \mathsf{C}'_p)$ sauf C'_j
- la contrainte résultant de la collision de C_i avec C'_j.

Nettoyage par test d'inclusion

Un nettoyage très simple des ensembles de contraintes contenant des disjonctions peut être effectué à n'importe quel moment (y compris en dehors des points de généralisation) : lorsqu'un ensemble de contraintes Φ contient deux disjonctions $(\mathsf{C}_1 \vee \cdots \vee \mathsf{C}_n)$ et $(\mathsf{C}'_1 \vee \cdots \vee \mathsf{C}'_p)$, si la première disjonction est « incluse » dans la seconde (c'est-à-dire si : $\forall i \in [1;n]\ .\ \exists j \in [1;p] \mid \mathsf{C}_i = \mathsf{C}'_j$), alors la seconde disjonction peut être supprimée de l'ensemble de contraintes.

D'un point de vue « logique », ce nettoyage est tout à fait justifié. En effet, si P et Q sont deux propriétés, nous avons bien l'équivalence :

$$(P \vee Q) \wedge P \;\Leftrightarrow\; P$$

Du point de vue de l'algorithme de saturation, si une disjonction de contraintes $(\mathsf{C}_1 \vee \cdots \vee \mathsf{C}_n)$ est incluse dans une autre disjonction de contraintes $(\mathsf{C}'_1 \vee \cdots \vee \mathsf{C}'_p)$, il est facile de vérifier que toute contrainte entrant en collision avec un membre de $(\mathsf{C}_1 \vee \cdots \vee \mathsf{C}_n)$ en engendrant une disjonction de contraintes ψ, entrera obligatoirement en collision avec un membre de $(\mathsf{C}'_1 \vee \cdots \vee \mathsf{C}'_p)$ en engendrant une disjonction de contraintes ψ' telle que ψ est inclus dans ψ'. De plus, toute collision avec un membre de $(\mathsf{C}'_1 \vee \cdots \vee \mathsf{C}'_p)$ qui n'est pas dans $(\mathsf{C}_1 \vee \cdots \vee \mathsf{C}_n)$ engendrera une disjonction de contraintes ψ' contenant $(\mathsf{C}_1 \vee \cdots \vee \mathsf{C}_n)$.

Par conséquent, pour toute disjonction de contraintes engendrée à partir de $(\mathsf{C}'_1 \vee \cdots \vee \mathsf{C}'_p)$, il existera systématiquement une disjonction de contraintes plus petite ou égale (au sens de l'inclusion) engendrée a partir de $(\mathsf{C}_1 \vee \cdots \vee \mathsf{C}_n)$. Comme un clash de typage ne peut être provoqué

que par l'obtention d'une disjonction vide, il n'y aura pas moins de clash si l'on supprime de l'ensemble de contraintes Φ la disjonction $(\mathsf{C}'_1 \vee \cdots \vee \mathsf{C}'_p)$, et tout programme refusé par le typeur sera donc toujours refusé après cette suppression. Un tel nettoyage peut donc être effectué sans risque d'invalider le système.

En pratique, la génération, lors de la saturation, de deux disjonctions incluses l'une dans l'autre n'est pas du tout anecdotique. Nous allons en particulier voir comment provoquer de telles situations pour profiter au maximum de cette technique de nettoyage.

Paresse ou exhaustivité ?

Pour chacune des fonctions LEFTS, RIGHTS, etc. que nous avons définies dans cette thèse, nous avons laissé la possibilité de considérer les disjonctions contenues dans l'ensemble de contraintes « à rotation près » ou pas. En d'autres termes, il existe deux implémentations possibles de ces fonctions. Elles peuvent :

- soit choisir un « représentant » de chacune des disjonctions et ne rechercher leur motif de contrainte que parmis ces représentants (nous nommerons cette implémentation « l'implémentation paresseuse »)
- soit rechercher leur motif de contrainte dans n'importe quel membre de n'importe quelle disjonction (nous nommerons cette implémentation « l'implémentation exhaustive »).

D'un point de vue logique, l'implémentation paresseuse est valide : pour que la conjonction complète soit « vraie », il suffit qu'un membre de chacune des disjonctions la composant soit vrai. Ces membres peuvent être choisis arbitrairement, la seule contrainte est qu'ils soient choisis de manière déterministe.

L'implémentation paresseuse peut a priori paraître plus intéressante puisque pour un ensemble de contraintes donné, le nombre de valeurs renvoyées par les fonctions LEFTS, RIGHTS, etc. sera inférieur ou égal au nombre de valeurs produites par l'implémentation exhaustive.

Pour une raison subtile, bien qu'elle puisse faire augmenter le nombre de contraintes considérées dans certaines situations, l'implémentation exhaustive donne de bien meilleures performances en pratique et des schémas de types plus lisibles que l'implémentation paresseuse.

Avec l'implémentation exhaustive, pour chaque disjonction de contraintes $(\mathsf{C}_1 \vee \cdots \vee \mathsf{C}_n)$ que l'on souhaite ajouter à un ensemble de contraintes Φ pour saturation, nous allons chercher, pour chaque C_i, l'ensemble de tous les C'_j de toutes les disjonctions de Φ entrant en collision avec C_i et générer les disjonctions engendrées par ces collisions. Bien évidemment, ceci ne revient pas à considérer que tous les membres de toutes les disjonctions sont vrais (et donc à considérer les disjonctions comme des conjonctions ce qui serait une catastrophe pour l'expressivité du système de types) puisque les disjonctions engendrées par chacune de ces collisions possèdent toujours les contraintes alternatives provenant des disjonctions originales.

L'intérêt de l'implémentation exhaustive est qu'elle va traquer les membres des disjonctions qui sont invalides ou incompatibles avec d'autres contraintes de Φ, et les supprimer le plus tôt

possible grâce à l'algorithme de nettoyage décrit dans la section précédente « **nettoyage par test d'inclusion** ». Elle va donc indirectement permettre de réduire les disjonctions manipulées lors de la saturation des contraintes.

Affinage de l'analyse de dépendances

Dans ce contexte particulier dans lequel les disjonctions ont toutes été considérées à rotation près, il est possible d'affiner encore plus le calcul de dépendances présenté dans la section 6.1.2.

Pour mettre en évidence notre objectif, considérons le schéma de type suivant :

$$[\ \forall \alpha\alpha' \,.\, \alpha \mid \texttt{int} \leqslant \alpha \ \wedge\ (\alpha' \,\#\, \texttt{S} \ \vee\ \texttt{string} \leqslant \alpha)\ ]$$

Si une contrainte (comme par exemple $(\alpha \leqslant \texttt{int})$) entre en collision avec $(\texttt{string} \leqslant \alpha)$ et engendre une contrainte invalide (dans notre exemple : $(\texttt{string} \leqslant \texttt{int})$), la saturation de contraintes engendrerait simplement la contrainte $(\alpha' \,\#\, \texttt{S})$ seule. Cependant, la variable de type α' est généralisée, il est alors impossible qu'une contrainte entre en collision avec $(\alpha' \,\#\, \texttt{S})$ et provoque un clash. En conséquence, nous pouvons supprimer de ce schéma de type la contrainte $(\alpha' \,\#\, \texttt{S} \ \vee\ \texttt{string} \leqslant \alpha)$ et la variable α' pour obtenir :

$$[\ \forall \alpha \,.\, \alpha \mid \texttt{int} \leqslant \alpha\ ]$$

qui est bien plus simple et plus lisible que le schéma original.

Heureusement, grâce à l'implémentation exhaustive, un tel nettoyage peut être effectué sans difficulté. En effet, pour nettoyer un schéma de type $[\ \forall \alpha_1 \ldots \alpha_n \,.\, \alpha_0 \mid \Phi_0\ ]$ dans lequel Φ_0 a été saturé grâce à l'implémentation exhaustive, il est possible de supprimer, de Φ_0, toutes les disjonctions de contraintes dont l'un des membres est indépendant (directement et indirectement) de α_0. Jusqu'à maintenant, pour être supprimées, il fallait que tous les membres soient indépendants de α_0, ce qui était beaucoup plus restrictif.

Nous remarquerons qu'une telle technique de nettoyage, beaucoup plus agressive, ne fonctionne que si l'ensemble de contraintes présent dans le schéma a été saturé en utilisant l'implémentation exhaustive.

Pour s'en convaincre, observons l'utilisation de cette technique de nettoyage sur le schéma de type :

$$[\ \forall \alpha\alpha' \,.\, \alpha \mid (\texttt{int} \leqslant \alpha \ \vee\ \alpha' \leqslant \texttt{bool}) \ \wedge\ (\alpha'' \,\#\, \texttt{S} \ \vee\ \texttt{string} \leqslant \alpha')\ ]$$

qui a été saturé avec l'implémentation paresseuse de la saturation en utilisant $(\texttt{int} \leqslant \alpha)$ et $(\alpha'' \,\#\, \texttt{S})$ comme représentants des disjonctions. Puisque la contrainte $(\alpha' \leqslant \texttt{bool})$ n'est pas liée à α et ne mentionne aucune variable non généralisée, le nettoyage comme nous l'avons décrit ici supprimerait la disjonction :

- $(\texttt{int} \leqslant \alpha \vee \alpha' \leqslant \texttt{bool})$

Il n'est néanmoins pas valide de supprimer cette disjonction puisqu'elle est susceptible d'être utilisée pour contraindre α. En effet, si la contrainte $(\texttt{S} \leqslant \alpha'')$ était générée par ailleurs (ce qui est possible puisque α'' n'est pas généralisée), la saturation engendrerait la contrainte $(\texttt{S} \,\#\, \texttt{S} \ \vee\ \texttt{string} \leqslant$

α') et donc la contrainte ($\texttt{string} \leqslant \alpha'$), qui, en entrant en collision avec ($\alpha' \leqslant \texttt{bool}$), générerait la disjonction ($\texttt{int} \leqslant \alpha \ \vee \ \texttt{string} \leqslant \texttt{bool}$) et donc ($\texttt{int} \leqslant \alpha$).

En revanche, la saturation exhaustive de l'ensemble de contraintes contenu dans le schéma :

$$[\, \forall \alpha\alpha' \,.\, \alpha \mid (\texttt{int} \leqslant \alpha \ \vee \ \alpha' \leqslant \texttt{bool}) \ \wedge \ (\alpha'' \,\#\, \texttt{S} \ \vee \ \texttt{string} \leqslant \alpha') \,]$$

engendre la disjonction :

$$\texttt{int} \leqslant \alpha \ \vee \ \texttt{string} \leqslant \texttt{bool} \ \vee \ \alpha'' \,\#\, \texttt{S}$$

et donc la disjonction :

$$\texttt{int} \leqslant \alpha \ \vee \ \alpha'' \,\#\, \texttt{S}$$

Nous obtenons alors le schéma de type :

$$\begin{aligned}[\, \forall \alpha\alpha' \,.\, \alpha \mid (\texttt{int} \leqslant \alpha \ \vee \ \alpha' \leqslant \texttt{bool}) \ \wedge \ (\alpha'' \,\#\, \texttt{S} \ \vee \ \texttt{string} \leqslant \alpha') \ \wedge \\ (\texttt{int} \leqslant \alpha \ \vee \ \texttt{string} \leqslant \texttt{bool} \ \vee \ \alpha'' \,\#\, \texttt{S}) \ \wedge \ (\texttt{int} \leqslant \alpha \ \vee \ \alpha'' \,\#\, \texttt{S}) \,]\end{aligned}$$

dans lequel les disjonctions :

- $\texttt{int} \leqslant \alpha \ \vee \ \alpha' \leqslant \texttt{bool}$
- $\alpha'' \,\#\, \texttt{S} \ \vee \ \texttt{string} \leqslant \alpha'$
- $\texttt{int} \leqslant \alpha \ \vee \ \texttt{string} \leqslant \texttt{bool} \ \vee \ \alpha'' \,\#\, \texttt{S}$

peuvent être supprimées par l'algorithme de nettoyage décrit dans cette section. Le schéma de type obtenu après saturation et nettoyage est alors :

$$[\, \forall \alpha \,.\, \alpha \mid (\texttt{int} \leqslant \alpha \ \vee \ \alpha'' \,\#\, \texttt{S}) \,]$$

qui est valide, et même plus petit que le schéma original !

Exemple

Pour mettre en évidence l'intérêt de considérer toutes les disjonctions de contraintes à rotation près, du nettoyage par test d'inclusion et de l'affinage du nettoyage grâce à l'implémentation exhaustive, comparons ces techniques avec l'approche paresseuse sur l'exemple de code suivant en utilisant le système du chapitre 3 :

```
let f = λ x . match x with
  ‖ I → 42
  ‖ S → "quarante deux" in
let n = f I in
...
```

Comme nous l'avons vu dans le chapitre 3 dans lequel cet exemple a déjà été étudié, le schéma de type inféré pour la fonction `f` est :

$$\begin{aligned}\texttt{f} : [\, \forall\, \alpha_1\, \alpha_2\, \alpha_3\ .\ \alpha_1 \mid\ & \\ & \alpha_2 \to \alpha_3 \leqslant \alpha_1 \\ \wedge\ & \alpha_2 \leqslant \{\, \texttt{I} \parallel \texttt{S} \,\} \\ \wedge\ & (\alpha_2 \,\#\, \texttt{I} \vee \texttt{int} \leqslant \alpha_3) \\ \wedge\ & (\alpha_2 \,\#\, \texttt{S} \vee \texttt{string} \leqslant \alpha_3)\,]\end{aligned}$$

Le typage de `(f I :` α`)` engendre alors en particulier les disjonctions de contraintes suivantes :

- $\alpha' \leqslant \{\, \texttt{I} \parallel \texttt{S} \,\}$
- $\alpha' \,\#\, \texttt{I} \vee \texttt{int} \leqslant \alpha$
- $\alpha' \,\#\, \texttt{S} \vee \texttt{string} \leqslant \alpha$
- $\texttt{I} \leqslant \alpha'$

liées les unes aux autres par les variables de type α et α'.

En utilisant l'implémentation paresseuse, des représentants de chacune des disjonctions sont choisis arbitrairement. Supposons qu'il s'agisse des contraintes suivantes :

- $\alpha' \leqslant \{\, \texttt{I} \parallel \texttt{S} \,\}$
- $\texttt{int} \leqslant \alpha$
- $\alpha' \,\#\, \texttt{S}$
- $\texttt{I} \leqslant \alpha'$

Seuls $(\alpha' \,\#\, \texttt{S})$ et $(\texttt{I} \leqslant \alpha')$ entrent en collision en engendrant la contrainte valide $(\texttt{I} \,\#\, \texttt{S})$. L'algorithme de saturation s'arrête alors très rapidement, en ne générant que la disjonction supplémentaire :

- $\texttt{I} \,\#\, \texttt{S} \vee \texttt{string} \leqslant \alpha$

dans laquelle nous supposons qu'il choisit arbitrairement pour représentant le membre $(\texttt{I} \,\#\, \texttt{S})$. Cet arrêt très rapide de la saturation pourrait passer naïvement pour un gain en performance. Cependant le schéma de type inféré pour `n` est alors assez « pollué » :

$$\texttt{n} : [\, \forall \alpha \alpha' \,.\, \alpha \mid \alpha' \leqslant \{\, \texttt{I} \parallel \texttt{S} \,\} \wedge (\alpha' \,\#\, \texttt{I} \vee \texttt{int} \leqslant \alpha) \wedge (\alpha' \,\#\, \texttt{S} \vee \texttt{string} \leqslant \alpha) \wedge (\texttt{I} \,\#\, \texttt{S} \vee \texttt{string} \leqslant \alpha) \wedge \texttt{I} \leqslant \alpha' \,]$$

et chaque utilisation de `n` dans la suite du code introduit des contraintes et des variables de type inutiles. Il est facile de voir que la variable `n` contient obligatoirement un entier. Il est assez aberrant de générer un schéma de type aussi compliqué pour représenter l'ensemble des entiers.

En revanche, si nous appliquons l'algorithme de saturation exhaustive sur l'ensemble de contraintes compris dans ce schéma, la disjonction $(\alpha' \,\#\, \texttt{I} \vee \texttt{int} \leqslant \alpha)$ entre en collision avec $(\texttt{I} \leqslant \alpha')$ et engendre la disjonction $(\texttt{I} \,\#\, \texttt{I} \vee \texttt{int} \leqslant \alpha)$. Comme $(\texttt{I} \,\#\, \texttt{I})$ est invalide, nous obtenons finalement la contrainte $(\texttt{int} \leqslant \alpha)$.

L'algorithme de nettoyage utilisant la version affinée de l'analyse de dépendances supprime alors toutes les disjonctions sauf $(\texttt{int} \leqslant \alpha)$. Le schéma de type associé à la variable `n` que nous obtenons finalement est bien :

$$[\, \forall \alpha \,.\, \alpha \mid \texttt{int} \leqslant \alpha \,]$$

6.1.4 Nettoyage par test de généralité

Motivation

La technique de nettoyage que nous allons présenter dans cette section est extrêmement puissante mais aussi très coûteuse en performance. Pour bien comprendre son utilité, commençons par observer le comportement de nos algorithmes de saturation sur un exemple utilisant un concept déjà intéressant en soi : la surcharge d'opérateurs.

Pour ce faire, nous nous plaçons dans le cadre d'un langage dont la bibliothèque d'exécution expose quelques primitives surchargées : les opérateurs $(+)$ et $(-)$ capables de prendre, soit deux entiers en argument et de renvoyer un entier, soit deux flottants et de renvoyer un flottant. Nous supposons en particulier que ces opérateurs ne sont pas capables de gérer deux opérandes de type différents (comme un entier et un flottant). Notre langage de contraintes nous permet parfaitement d'exprimer ce genre de propriété, il suffit d'associer à ces primitives le schéma de type :

$$\begin{aligned}[\ \forall\, \alpha\, \alpha_1\, \alpha_2\, \alpha_3\, \alpha'_1\, \alpha'_2\, \alpha'_3 \ .\ \alpha \ \mid \\ (\alpha_1 \to \alpha_2 \leqslant \alpha \ \vee\ \alpha'_1 \to \alpha'_2 \leqslant \alpha) \wedge \\ \alpha_1 \to \alpha_3 \leqslant \alpha_2 \ \wedge\ \alpha_1 \leqslant \texttt{int} \ \wedge\ \texttt{int} \leqslant \alpha_3 \wedge \\ \alpha'_1 \to \alpha'_3 \leqslant \alpha'_2 \ \wedge\ \alpha'_1 \leqslant \texttt{float} \ \wedge\ \texttt{float} \leqslant \alpha'_3 \ ]\end{aligned}$$

Ce schéma de type est, certes, un peu « compliqué » puisqu'il est sous forme normale. Avec un peu de sucre syntaxique, il est parfaitement possible de l'écrire plus simplement :

$$\texttt{int} \to \texttt{int} \to \texttt{int} \mid \texttt{float} \to \texttt{float} \to \texttt{float}$$

Une implémentation concrète de l'un de nos systèmes de types fournira bien entendu le préprocesseur et le pretty-printer qui vont bien pour lire et écrire les schémas de type de manière agréable pour un être humain. Nous ne nous intéressons pas spécialement à ce détail dans cette thèse.

Quoi qu'il en soit, l'ensemble de contraintes présent dans ce schéma peut se décomposer en trois parties :

- La disjonction $(\alpha_1 \to \alpha_2 \leqslant \alpha \ \vee\ \alpha'_1 \to \alpha'_2 \leqslant \alpha)$ spécifie que la variable de type α exposée par le schéma de type est associée soit à $(\alpha_1 \to \alpha_2)$, soit à $(\alpha'_1 \to \alpha'_2)$.
- Les trois contraintes $(\alpha_1 \to \alpha_3 \leqslant \alpha_2)$, $(\alpha_1 \leqslant \texttt{int})$ et $(\texttt{int} \leqslant \alpha_3)$ encodent le fait que $(\alpha_1 \to \alpha_2)$ représente le « type » $(\texttt{int} \to \texttt{int} \to \texttt{int})$.
- Les trois dernières contraintes $(\alpha'_1 \to \alpha'_3 \leqslant \alpha'_2)$, $(\alpha'_1 \leqslant \texttt{int})$ et $(\texttt{int} \leqslant \alpha'_3)$ encodent quant à elles le fait que $(\alpha'_1 \to \alpha'_2)$ représente le « type » $(\texttt{float} \to \texttt{float} \to \texttt{float})$.

Considérons maintenant le code suivant :

```
let f =
  if rand () then (λ x y → x + y)
  else (λ x y → x - y)
)
```

où la fonction `rand` renvoie un booléen aléatoire.

L'utilisation de l'un de nos systèmes sur ce code inférera pour `f` le schéma de type qui, après nettoyage avec les techniques présentées précédemment, sera :

$$
\begin{aligned}
[\ \forall\, \alpha\, \alpha_1\, \alpha_2\, \alpha_3\, \alpha'_1\, \alpha'_2\, \alpha'_3 \alpha''_1, \alpha''_2, \alpha''_3, \alpha'''_1, \alpha'''_2, \alpha'''_3\ .\ \alpha\ \mid\ & \\
& (\alpha_1 \to \alpha_2 \leqslant \alpha\ \vee\ \alpha'_1 \to \alpha'_2 \leqslant \alpha) \wedge \\
& \alpha_1 \to \alpha_3 \leqslant \alpha_2\ \wedge\ \alpha_1 \leqslant \texttt{int}\ \wedge\ \texttt{int} \leqslant \alpha_3 \wedge \\
& \alpha'_1 \to \alpha'_3 \leqslant \alpha'_2\ \wedge\ \alpha'_1 \leqslant \texttt{float}\ \wedge\ \texttt{float} \leqslant \alpha'_3 \wedge \\
& (\alpha''_1 \to \alpha''_2 \leqslant \alpha\ \vee\ \alpha'''_1 \to \alpha'''_2 \leqslant \alpha) \wedge \\
& \alpha''_1 \to \alpha''_3 \leqslant \alpha''_2\ \wedge\ \alpha''_1 \leqslant \texttt{int}\ \wedge\ \texttt{int} \leqslant \alpha''_3 \wedge \\
& \alpha'''_1 \to \alpha'''_3 \leqslant \alpha'''_2\ \wedge\ \alpha'''_1 \leqslant \texttt{float}\ \wedge\ \texttt{float} \leqslant \alpha'''_3\ ]
\end{aligned}
$$

Ce schéma est excessivement « gros », on aurait pu s'attendre à obtenir le même schéma que pour les primitives $(+)$ et $(-)$. En réalité, le problème vient du fait que les deux utilisations indépendantes des opérateurs $(+)$ et $(-)$ ont engendré deux instanciations indépendantes de leur schéma de type associé. Ceci a engendré une duplication des contraintes avec des variables de type différentes.

Comme on peut s'y attendre, en plus de ne pas être pratique à lire pour le programmeur, de telles réplications de contraintes peuvent poser de graves problèmes de performance.

Pour les supprimer, une première idée est de chercher à repérer des sous-ensembles de contraintes « égaux à renommage près des variables de type ». Cette technique peut fonctionner dans certaines situations, comme par exemple celle que nous avons décrite ci dessus, mais est particulièrement coûteuse, et moins générale qu'on pourrait l'espérer.

Il est en effet assez courant que deux sous-ensembles de contraintes ne soient pas parfaitement égaux à renommage près. Il peut arriver qu'ils expriment la même « propriété » mais avec des contraintes écrites d'une autre manière. À ce sujet, nous remarquerons que le schéma de type que nous avons défini pour les opérateurs surchargés $(+)$ et $(-)$ est loin d'être le seul possible. Il lui en existe de nombreux équivalents comme par exemple :

$$
\begin{aligned}
[\ \forall\, \alpha\, \alpha_1\, \alpha_2\, \alpha_3\ .\ \alpha\ \mid\ & \\
& \alpha_1 \to \alpha_2 \leqslant \alpha\ \wedge\ \alpha_1 \to \alpha_3 \leqslant \alpha_2 \wedge \\
& (\alpha_1 \leqslant \texttt{int}\ \vee\ \alpha_1 \leqslant \texttt{float}) \wedge \\
& (\alpha_1 \not\leqslant \texttt{int}\ \vee\ \texttt{int} \leqslant \alpha_3) \wedge \\
& (\alpha_1 \not\leqslant \texttt{float}\ \vee\ \texttt{float} \leqslant \alpha_3)\ ]
\end{aligned}
$$

dans lequel :

- Les contraintes $(\alpha_1 \to \alpha_2 \leqslant \alpha)$ et $(\alpha_1 \to \alpha_3 \leqslant \alpha_2)$ spécifient que la variable α exposée par le schéma est une fonction à deux arguments. Le type des arguments est représenté par α_1 et le type du résultat par α_3.
- La disjonction $(\alpha_1 \leqslant \texttt{int}\ \vee\ \alpha_1 \leqslant \texttt{float})$ impose les arguments à être soit des entiers soit des flottants.

- La disjonction $(\alpha_1 \not\leqslant \texttt{int} \ \vee \ \texttt{int} \leqslant \alpha_3)$ contraint le résultat à être un entier si les arguments sont des entiers.
- La disjonction $(\alpha_1 \not\leqslant \texttt{float} \ \vee \ \texttt{float} \leqslant \alpha_3)$ contraint le résultat à être un flottant si les arguments sont des flottants.

Il n'est a priori pas du tout « évident » que ce schéma est « équivalent » à celui que nous avons donné tout à l'heure pour les opérateurs $(+)$ et $(-)$. Dans la pratique, il arrive qu'une même variable de type soit, dans un schéma, contrainte de deux manières différentes mais équivalentes. Il serait alors intéressant de repérer automatiquement une telle situation et de supprimer l'un des deux sous-ensembles de contraintes équivalents.

Les types existentiels à notre secours

Il existe en fait une manière très simple mais particulièrement coûteuse de nettoyer de tels schémas : il suffit de tester, pour chaque disjonction, si sa suppression engendre un schéma de type strictement plus général ou pas. Un tel test de généralité peut être effectué grâce aux types existentiels comme présentés dans la section 5.6 du chapitre sur les **GADT**.

Le test de généralité étant assez coûteux en pratique, il est rentable d'exécuter les autres algorithmes de nettoyage à chaque détection et suppression d'une disjonction inutile. Ceci permet en effet de supprimer beaucoup plus rapidement tout un sous-ensemble de disjonctions lorsque les liens les reliant sont cassés par la suppression de l'une d'entre elles.

Le problème de cette technique de nettoyage (en dehors de ses performances, qui restent néanmoins acceptables la plupart du temps) est que les sous-ensembles de contraintes qui sont supprimés dépendent de l'ordre dans lequel on a testé les disjonctions. Il est possible d'affiner cette technique pour supprimer les sous-ensembles de contraintes donnant le schéma de type le plus « petit » au final (la définition de « plus petit » étant un peu arbitraire : on peut chercher à minimiser le nombre de disjonctions, le nombre de disjonctions non-réduites à un élément, mettre des poids différents sur les différentes constructions de type, etc.), mais ce genre d'amélioration est peu utile en pratique et encore plus coûteuse. La version de base fonctionne déjà plutôt bien.

6.2 Une structure de données dédiée aux ensembles de contraintes

6.2.1 Motivation

Les algorithmes de saturation et de nettoyage nécessitent beaucoup d'accès aux ensembles de contraintes, en particulier via les fonctions LEFTS, RIGHTS, etc. Il peut être intéressant de les représenter sous une forme optimisée pour ces accès.

De plus, il est assez courant, en pratique, d'obtenir d'assez longues « chaînes » de variables de type reliées par des $(\leqslant)$, c'est-à-dire de la forme $(\alpha_1 \leqslant \alpha_2 \leqslant \cdots \leqslant \alpha_n)$ avec n « assez grand ». En plus d'être liées entre elles, chacune de ces variables est liée avec d'éventuels types construits par des

relations de sous-typage et de non-sous-typage. Les algorithmes de saturation comme nous les avons décrits vont alors construire toutes les contraintes de la forme $(\alpha_i \leqslant \alpha_j)$ avec $(1 \leqslant i < j \leqslant n)$ qui sont en nombre quadratique par rapport à n. Ainsi, lors de l'arrivée d'une nouvelle contrainte de la forme $(\tau^l \leqslant \alpha_1)$, l'algorithme de saturation naïf va chercher tous les τ^r plus grands que α_1 et générer toutes les contraintes $(\tau^l \leqslant \tau^r)$ donc en particulier toutes les contraintes de la forme $(\tau^l \leqslant \alpha_i)$ avec $(2 \leqslant i \leqslant n)$. Chacune de ces contraintes va alors générer en cascade de multiples comparaisons redondantes entre τ^l et les différents α_i. Bien entendu, grâce à la mémoïsation, le mécanisme de calcul de point fixe va arrêter le calcul dès qu'il obtient une contrainte qu'il a déjà rencontrée, mais il est assez clair qu'une représentation des ensembles de contraintes sous forme d'une liste de disjonctions est loin d'être optimale en occupation mémoire et ne favorise pas le temps de calcul lors de la saturation.

6.2.2 Idée

Une idée pour améliorer les performances consiste à représenter les ensembles de contraintes sous forme d'un graphe orienté mutable dont :

- les noeuds sont des types,
- les arêtes représentent une relation de sous-typage ou de non-sous-typage entre deux types,
- les arêtes possèdent un attribut représentant la liste des ensembles de contraintes alternatives à cette relation (c'est-à-dire la liste dont chaque élément est la liste des autres membres d'une disjonction dans laquelle se trouve cette relation).

Nous ne représentons dans ce graphe que les disjonctions ne contenant que des contraintes reliant une variable de type avec un type construit (c'est-à-dire un type qui n'est pas une variable de type) ou deux variables de type entre elles. En effet, une contrainte reliant deux types construits n'est jamais utilisée par les fonctions LEFTS, RIGHTS, etc.

Par ailleurs, la relation de sous-typage étant transitive, nous ne représentons pas, dans ce graphe, les relations de sous-typage pouvant se déduire d'un chemin composé de relations de sous-typage dans le même sens.

De manière similaire, nous ne représentons pas les relations de non-sous-typage pouvant se déduire d'un chemin composé d'une relation de non-sous-typage dans le même sens et d'un nombre quelconque de relations de sous-typage en sens inverse.

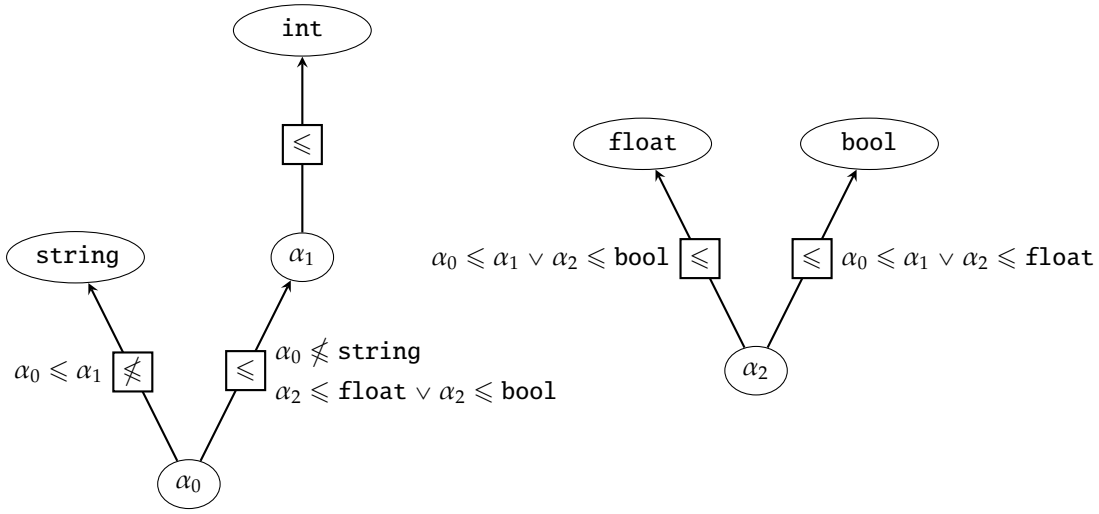
Pour encoder les fonctions LEFTS, RIGHTS, etc., et extraire les types associés à une variable de type donnée en argument, il suffit alors de parcourir les noeuds représentant des variables de type du graphe en partant de la variable donnée, et d'agréger tous les types liés à cette variable par la relation qui nous intéresse.

6.2.3 Exemple

Considérons l'ensemble de contraintes Φ calculé par saturation des trois disjonctions suivantes grâce au système du chapitre 5 :

- $\alpha_0 \leqslant \alpha_1 \ \vee \ \alpha_0 \not\leqslant \texttt{string}$
- $\alpha_0 \leqslant \alpha_1 \ \vee \ \alpha_2 \leqslant \texttt{float} \ \vee \ \alpha_2 \leqslant \texttt{bool}$
- $\alpha_1 \leqslant \texttt{int}$

Le graphe permettant de représenter Φ est le suivant :



Nous remarquerons en particulier que la contrainte $(\alpha_0 \leqslant \texttt{int} \vee \alpha_2 \leqslant \texttt{float} \vee \alpha_2 \leqslant \texttt{bool})$, que l'on obtient par saturation, n'est représentée ni par une arête liant α_0 à `int`, ni par un attribut sur l'arête liant α_2 à `float`. Cette disjonction peut en effet se déduire de l'arête liant α_0 à α_1 et de celle liant α_1 à `int`.

Pour calculer $\textsc{rights}(\alpha_0, \Phi)$, et donc extraire toutes les disjonctions contenant une contrainte de la forme $(\alpha_0 \leqslant \tau^r)$, il suffit de parcourir ce graphe à partir du noeud α_0 en suivant les arête annotées par un $(\leqslant)$. On obtient alors les quatre disjonctions suivantes :

- $\alpha_0 \leqslant \alpha_1 \vee \alpha_0 \not\leqslant \texttt{string}$
- $\alpha_0 \leqslant \alpha_1 \vee \alpha_2 \leqslant \texttt{float} \vee \alpha_2 \leqslant \texttt{bool}$
- $\alpha_0 \leqslant \texttt{int} \vee \alpha_0 \not\leqslant \texttt{string}$
- $\alpha_0 \leqslant \texttt{int} \vee \alpha_2 \leqslant \texttt{float} \vee \alpha_2 \leqslant \texttt{bool}$

6.2.4 Remarques

Il est possible d'améliorer les performances de l'algorithme de saturation en encodant les fonctions LEFTS et RIGHTS pour qu'elles n'extraient de l'ensemble de contraintes que les types construits et qu'elles ignorent les variables de type universelles intermédiaires. Il faut néanmoins faire un cas particulier lorsque la comparaison que l'on souhaite ajouter concerne un type existentiel. En

effet, une relation entre un type existentiel et une variable universelle peut engendrer d'autres contraintes lorsque la variable universelle n'est pas dans la portée du type existentiel. Dans cette situation, il faut extraire de l'ensemble de contraintes tous les types reliés à la variable universelle donnée par la relation qui nous intéresse, y compris les autres variables universelles.

Pour favoriser la mémoïsation, il pourrait paraître intéressant de stocker dans une seconde structure de données (permettant typiquement un test d'appartenance rapide comme une table de hachage ou un arbre binaire de recherche équilibré), l'ensemble des contraintes ayant déjà été rencontrées, y compris les relations entre deux types construits. Cette optimisation a été testée mais ne s'est pas montrée rentable sur la plupart de nos exemples. La raison est probablement la suivante : comme les types construits ne sont pas arborescents chez nous (à l'exception des types existentiels), une comparaison entre deux types construits se transforme très rapidement en des comparaisons entre des variables de type. La mémoïsation possible grâce au graphe, concernant uniquement les contraintes reliant une variable de type avec un type construit ou avec une autre variable de type, permet donc très rapidement d'arrêter la saturation, même lors de la rencontre d'une relation entre deux types construits ayant déjà été rencontrée auparavant. En revanche, la structure stockant toutes les relations rencontrées a tendance à grossir rapidement et y accéder en plus d'accéder au graphe fait statistiquement plus perdre que l'amélioration de la mémoïsation ne nous fait gagner.

6.2.5 Algorithme impératif de saturation

Pour mettre en évidence l'utilisation d'un tel graphe pour représenter les ensembles de contraintes, nous exprimons ici de manière impérative l'algorithme de saturation.

Le typage d'un programme débute avec un graphe vide représentant un ensemble de contraintes vide. À chaque application d'une règle de typage, de nouvelles disjonctions de contraintes sont générées les unes après les autres. Pour gérer une nouvelle disjonction ψ, on commence par tester si elle contient une contrainte liant deux types construits (comme par exemple $(\alpha_1 \to \alpha_2 \leqslant \alpha_3 \to \alpha_4)$). Dans ce cas on applique la règle de saturation correspondante sur cette contrainte, conduisant en général à la propagation de la relation sur les membres des types construits en question. Si ψ est vide, le typage s'arrête sur une erreur (il est alors important d'avoir transporté des informations de localisation pour générer un message correct).

Lorsque tous les membres de ψ relient une variable de type avec un type construit ou avec une autre variable de type, on l'insère dans le graphe. Pour ce faire, il suffit de suivre les étapes suivantes :

1. On teste si une disjonction plus ou aussi « forte » que ψ n'est pas déjà présente : pour chaque membre de la disjonction ψ, reliant obligatoirement une variable de type à un autre type, on parcourt le graphe en partant de cette variable à la recherche de ψ ou d'une disjonction incluse dans ψ. Si une telle disjonction plus forte est trouvée, il n'est pas nécessaire d'insérer ψ, on s'arrête. Nous remarquerons que pour éviter l'insertion d'un ψ inutile, il n'est pas suffisant d'effectuer ce test sur uniquement l'un des membres de ψ.

2. Si ψ contient une relation entre un type existentiel et une variable universelle hors de sa portée, on applique la règle de saturation correspondante générant de nouvelles disjonctions.
3. On insère toutes les rotations de ψ dans le graphe.
4. On supprime du graphe toutes les disjonctions moins fortes que ψ (c'est-à-dire incluant ψ).
5. On extrait de Φ toutes les disjonctions entrant en collision avec ψ, et on applique les règles de saturation correspondantes générant d'éventuelles nouvelles disjonctions.

Lors des différents parcours du graphe, à savoir :

- la recherche d'une disjonction plus « forte »,
- la suppression des disjonctions moins « fortes » et
- la recherche de toutes les disjonctions liées à une variable de type,

il faut prendre garde à ne pas boucler quand des variables de type sont liées par un cycle de relations de sous-typage comme $(\alpha_1 \leqslant \alpha_2 \leqslant \alpha_3 \leqslant \alpha_1)$. Pour ce faire, il suffit d'utiliser un simple algorithme de coloration des arêtes du graphe permettant de savoir par où les algorithmes de parcours sont déjà passés.

6.2.6 Implémentation

Dans nos implémentations, les variables de type universelles sont simplement représentées par des entiers générés dans l'ordre à partir de 0. La structure de graphe représentant les ensembles de contraintes est simplement encodé par un tableau, grossissant à la demande, et dont la case d'indice i est associée à la variable de type représentée par l'entier i. Chaque case de ce tableau stocke alors huit listes de couples :

- La liste des variables de type plus petites associées aux contraintes alternatives provenant des disjonctions dont cette relation provient.
- La liste des types construits plus petits associés aux contraintes alternatives correspondantes.
- La liste des variables de type qui ne sont pas plus petites associées aux contraintes alternatives correspondantes.
- etc.

Un tel encodage permet d'accéder très rapidement à la liste des contraintes associées à une variable de type.

Un inconvénient de cet encodage est que les variables de type qui ne sont plus utilisées et les contraintes qui leur sont associées ne sont pas supprimées par le ramasse miettes. Néanmoins, à l'exception des règles du chapitre 4, seules les règles de typage et d'instanciation génèrent de nouvelles variables de type universelles. Cependant, le nombre de variables de type générées par les règles de typage est proportionnel à la taille du programme, et grâce aux algorithmes de nettoyage, les schémas de type grossissent lentement ce qui limite le nombre de variables fraîches générées à chaque instanciation.

À titre informatif, voici les caractéristiques de l'implémentation du système du chapitre 5 utilisant cette représentation des ensembles de contraintes :

- Le code complet fait environ 3200 lignes d'OCaml dont :
  - Environ 600 lignes pour le frontend du langage.
  - Environ 700 lignes pour l'encodage des règles de typage et d'instanciation.
  - Environ 400 lignes pour l'encodage de la structure de graphe et des primitives associées.
  - Environ 700 lignes pour l'algorithme de saturation.
  - Environ 200 lignes pour l'évaluateur.
  - Le reste est composé du programme principal et d'utilitaires de test et de benchmarks.

6.3 Distribution du code

Les différents systèmes de types de cette thèse ont été implémentés et testés. Les techniques d'optimisation décrites dans ce chapitre ont été en particulier expérimentées sur le système du chapitre 5 concernant les `GADT`. L'implémentation correspondante est disponible à l'adresse :

`https://github.com/bvaugon/GADT-implem`

Après avoir cloné le dépôt, le programme peut être compilé grâce à la commande :

```
$ make
```

Un top-level peut être obtenu grâce à la commande :

```
$ make top
```

Il est également possible de lancer les tests contenus dans les différents fichiers du dossier `tests/` grâce à la commande :

```
$ make tests:<nom-du-test>
```

Pour chaque expression, le programme affiche trois informations :

- `Expr:` l'expression originale pretty-printée après parsing.
- `Eval:` le résultat de l'évaluation de l'expression.
- `Type:` le type inféré pour l'expression.

Conclusion

Dans cette thèse, nous nous sommes intéressés à la vérification de validité des programmes par typage. Plus précisément, nous nous sommes concentrés sur le domaine du sous-typage et de la synthèse de types.

Après avoir introduit le langage sur lequel nous travaillons (un ML étendu) et sa sémantique, nous avons présenté un formalisme adapté à notre problématique. Celui-ci permet d'exprimer nos systèmes de types grâce à un unique ensemble de règles d'inférence, spécifiant, de manière uniforme, comment extraire des contraintes de sous-typage d'un programme et vérifier leur cohérence.

Dans ce formalisme, nous avons défini un premier système de types servant de base au reste de la thèse. Nous avons alors prouvé la correction de ce système vis-à-vis de la sémantique de notre langage, et démontré la terminaison de l'algorithme d'inférence associé.

Nous avons ensuite étendu ce système de base dans trois directions orthogonales. La première extension présentée permet d'affiner le typage du filtrage de motifs en associant, pour chaque cas de filtrage, des informations reliant le motif aux contraintes de types générées lors du typage du corps de ce cas. La deuxième extension présentée permet d'étendre le mécanisme de généralisation standard de ML. Utilisée conjointement avec la première, elle permet d'encoder une forme de « programmation objet » dans le langage lui-même, sans construction supplémentaire. La dernière extension formalise une version originale de « type existentiel », permettant de définir des `GADT` adaptés à notre contexte de sous-typage. Notre encodage des types existentiels, en plus de sa bonne intégration avec le sous-typage, permet de pousser l'inférence de type bien plus loin qu'avec les `GADT` classiques.

Nous avons alors montré comment gérer proprement le problème de la récursion polymorphe en présence de sous-typage en utilisant les mécanismes initialement introduits pour définir les `GADT`.

Pour chacune de nos extensions, nous avons montré comment adapter les preuves de terminaison et de correction du système de base.

Enfin, nous avons montré comment implémenter, de manière performante, les systèmes de types décrits dans cette thèse. Ces techniques d'implémentation ont été testées et semblent bien fonctionner en pratique.

Ce travail ouvre la porte à différents travaux. En premier lieu, les preuves de validité de nos systèmes sont assez volumineuses et ont été entièrement faites à la main. Pour plus de sûreté, il serait intéressant d'encoder nos systèmes et les preuves de validité associées en langage Coq.

Par ailleurs, dans le contexte de la conception d'un « vrai » langage de programmation, il serait intéressant d'offrir une syntaxe concrète des schémas de type pour qu'ils soient plus simples à utiliser pour le programmeur. En présence de disjonctions et de négations dans les ensembles de contraintes, cette tâche risque d'être particulièrement difficile. Un tel travail semble avoir peu d'intérêt théorique, mais pourrait être très utile en pratique.

Enfin, le formalisme que nous avons introduit pour définir nos systèmes présente l'avantage notable de permettre de les implémenter automatiquement. Une telle implémentation n'est en général pas très performante et différentes optimisations, comme celles que nous avons présentées dans cette thèse, sont nécessaires pour son passage à l'échelle. Néanmoins, un compilateur prenant en entrée un ensemble de règles et générant un typeur serait fort utile pour mettre au point et prototyper des systèmes de types écrits dans notre formalisme. Un tel typeur, généré à partir d'un ensemble de règles, pourrait même produire automatiquement l'arbre d'inférence associé à un programme, ce qui serait très pratique pour la mise au point.

Bibliographie

[AW94] Alexander Aiken, Edward L. Wimmers et T. K. Lakshman. « Soft Typing with Conditional Types ». In : POPL '94 (Principle of Programming Languages) (1994), p. 163–173. doi : 10.1145/174675.177847. url : http://doi.acm.org/10.1145/174675.177847 (cf. page 93).

[CF05] Giuseppe Castagna et Alain Frisch. « A Gentle Introduction to Semantic Subtyping ». In : PPDP '05 (International Conference on Principles and Practice of Declarative Programming). Lisbon, Portugal : ACM, 2005, p. 198–199. isbn : 1-59593-090-6. doi : 10.1145/1069774.1069793. url : http://doi.acm.org/10.1145/1069774.1069793 (cf. page 28).

[CP15] Arthur Charguéraud et François Pottier. « Machine-Checked Verification of the Correctness and Amortized Complexity of an Efficient Union-Find Implementation ». In : ITP '15 (Interactive Theorem Proving) 9236 (août 2015), p. 137–153 (cf. pages 6, 27, 195).

[CD86] Dominique Clément, Thierry Despeyroux, Gilles Kahn et Joëlle Despeyroux. « A Simple Applicative Language : Mini-ML ». In : LFP '86 (LISP Functional Programming) (1986), p. 13–27. doi : 10.1145/319838.319847. url : http://doi.acm.org/10.1145/319838.319847 (cf. pages 6, 9).

[CC77] Patrick Cousot et Radhia Cousot. « Abstract Interpretation : A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints ». In : POPL '77 (Principle of Programming Languages) (1977), p. 238–252. doi : 10.1145/512950.512973. url : http://doi.acm.org/10.1145/512950.512973 (cf. page 6).

[CR14] Julien Cretin et Didier Rémy. « System F with Coercion Constraints ». In : LICS '14 (Logics in Computer Science). ACM, juil. 2014 (cf. page 122).

[DM82] Luis Damas et Robin Milner. « Principal Type-schemes for Functional Programs ». In : POPL '82 (Principle of Programming Languages) (1982), p. 207–212. doi : 10.1145/582153.582176. url : http://doi.acm.org/10.1145/582153.582176 (cf. pages 8, 13, 14, 121).

[DM15] Stephen Dolan et Alan Mycroft. « Polymorphism, subtyping and type inference in MLsub ». In : Computer (2015) (cf. page 28).

[ES95] Jonathan Eifrig, Scott Smith et Valery Trifonov. « Sound Polymorphic Type Inference for Objects ». In : SIGPLAN Notices 30.10 (oct. 1995), p. 169–184. issn : 0362-1340. doi : 10.1145/217839.217858. url : http://doi.acm.org/10.1145/217839.217858 (cf. page 29).

[FC08] Alain Frisch, Giuseppe Castagna et Véronique Benzaken. « Semantic Subtyping : Dealing Set-theoretically with Function, Union, Intersection, and Negation Types ». In : JACM '08 (Journal of the ACM) 55.4 (sept. 2008), 19 :1–19 :64. issn : 0004-5411. doi : 10.1145/1391289.1391293. url : http://doi.acm.org/10.1145/1391289.1391293 (cf. page 28).

[G04^2] Jacques Garrigue. « Relaxing the Value Restriction ». In : FLOPS '04 (Functional and Logic Programming). 2004, p. 196–213. doi : 10.1007/978-3-540-24754-8_15. url : http://dx.doi.org/10.1007/978-3-540-24754-8_15 (cf. page 82).

[G02] Jacques Garrigue. « Simple Type Inference for Structural Polymorphism ». In : FOOL '02 (Foundations of Object-Oriented Languages) (2002) (cf. page 93).

[G04^1] Jacques Garrigue. « Typing deep pattern-matching in presence of polymorphic variants ». In : JSSST '04 (Workshop on Programming and Programming Languages) (2004) (cf. page 123).

[GL11] Jacques Garrigue et Jacques Le Normand. « Adding GADTs to OCaml : a direct approach ». In : ML '11 (Workshop on ML and its applications) (2011) (cf. page 148).

[G86] Jean-Yves GIRARD. « The system F of variable types, fifteen years later ». In : Theoretical Computer Science (1986), p. 159–192 (cf. page 122).

[G11] Eli Gottlieb. « Simple, Decidable Type Inference with Subtyping ». In : CoRR '11 (Computing Research Repository - arXiv) abs/1104.3116 (2011). url : http://arxiv.org/abs/1104.3116 (cf. page 28).

[H91] Fritz Henglein. « Efficient Type Inference for Higher-Order Binding-Time Analysis ». In : FPCA '91 (Functional Programming and Computer Architecture) (1991), p. 448–472 (cf. pages 6, 27, 195).

[HP01] Haruo Hosoya et Benjamin Pierce. « Regular Expression Pattern Matching for XML ». In : SIGPLAN Notices 36.3 (jan. 2001), p. 67–80. issn : 0362-1340. doi : 10.1145/373243.360209. url : http://doi.acm.org/10.1145/373243.360209 (cf. page 28).

[HP03] Haruo Hosoya et Benjamin C. Pierce. « XDuce : A Statically Typed XML Processing Language ». In : TOIT '03 (Transactions on Internet Technology) 3.2 (mai 2003), p. 117–148. issn : 1533-5399. doi : 10.1145/767193.767195. url : http://doi.acm.org/10.1145/767193.767195 (cf. page 28).

[KW99] Assaf J. Kfoury et Joe B. Wells. « Principality and Decidable Type Inference for Finite-rank Intersection Types ». In : POPL '99 (Principle of Programming Languages) (1999), p. 161–174. doi : 10.1145/292540.292556. url : http://doi.acm.org/10.1145/292540.292556 (cf. pages 122, 123).

[L08] Daan Leijen. « HMF : Simple Type Inference for First-class Polymorphism ». In : SIGPLAN Notices 43.9 (sept. 2008), p. 283–294. issn : 0362-1340. doi : 10.1145/1411203.1411245. url : http://doi.acm.org/10.1145/1411203.1411245 (cf. page 122).

[LW91] Xavier Leroy et Pierre Weis. « Polymorphic Type Inference and Assignment ». In : POPL '91 (Principle of Programming Languages) (1991), p. 291–302. doi : 10.1145/99583.99622. url : http://doi.acm.org/10.1145/99583.99622 (cf. page 82).

[LD14] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy et Jérôme Vouillon. *The OCaml system (release 4.02) : Documentation and user's manual*. Inria. Sept. 2014. url : http://caml.inria.fr/pub/docs/manual-ocaml/ (cf. pages 8, 91).

[LO92] Konstantin Läufer et Martin Odersky. « An Extension of ML with First-Class Abstract Types ». In : ML '92 (Workshop on ML and its applications) (1992) (cf. page 148).

[MM82] Alberto Martelli et Ugo Montanari. « An Efficient Unification Algorithm ». In : TOPLAS '82 (Transactions on Programming Languages and Systems) 4.2 (avr. 1982), p. 258–282. issn : 0164-0925. doi : 10.1145/357162.357169. url : http://doi.acm.org/10.1145/357162.357169 (cf. page 6).

[MP94] Michel Mauny et François Pottier. *An Implementation of Caml-Light with existential types*. Research Report RR-2183. INRIA, 1994. url : https://hal.inria.fr/inria-00074488 (cf. page 148).

[PJ06] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich et Geoffrey Washburn. « Simple Unification-based Type Inference for GADTs ». In : SIGPLAN Notices 41.9 (sept. 2006), p. 50–61. issn : 0362-1340. doi : 10.1145/1160074.1159811. url : http://doi.acm.org/10.1145/1160074.1159811 (cf. page 6).

[P91] Benjamin C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. Rapp. tech. 1991 (cf. page 122).

[P00] François Pottier. « A Versatile Constraint-Based Type Inference System ». In : Nordic Journal of Computing 7.4 (nov. 2000), p. 312–347. url : http://gallium.inria.fr/~fpottier/publis/fpottier-njc-2000.ps.gz (cf. page 93).

[P01] François Pottier. « Simplifying subtyping constraints : a theory ». In : Information & Computation 170.2 (nov. 2001), p. 153–183. url : http://gallium.inria.fr/~fpottier/publis/fpottier-ic01.ps.gz (cf. page 28).

[PR06] François Pottier et Yann Régis-Gianas. « Stratified type inference for generalized algebraic data types ». In : POPL '06 (Principle of Programming Languages) (jan. 2006), p. 232–244. doi : http://doi.acm.org/10.1145/1111037.1111058. url : http://gallium.inria.fr/~fpottier/publis/pottier-regis-gianas-popl06.ps.gz (cf. page 148).

[PR05] François Pottier et Didier Rémy. « Advanced Topics in Types and Programming Languages ». In : MIT Press, 2005. Chap. The Essence of ML Type Inference (cf. page 6).

[R95] Didier Rémy. « A case study of typechecking with constrained types : Typing record concatenation ». 1995 (cf. page 93).

[SR13] Gabriel Scherer et Didier Rémy. « GADTs Meet Subtyping ». In : ESOP '13 (European Conference on Programming Languages and Systems). Rome, Italy : Springer-Verlag, 2013, p. 554–573. isbn : 978-3-642-37035-9. doi : 10.1007/978-3-642-37036-6_30 (cf. page 148).

[SP09] Tom Schrijvers, Simon Peyton Jones, Martin Sulzmann et Dimitrios Vytiniotis. « Complete and Decidable Type Inference for GADTs ». In : SIGPLAN Notices 44.9 (août 2009), p. 341–352. issn : 0362-1340. doi : 10.1145/1631687.1596599. url : http://doi.acm.org/10.1145/1631687.1596599 (cf. page 148).

[SP05] Vincent Simonet et François Pottier. *Constraint-Based Type Inference for Guarded Algebraic Data Types*. Research Report 5462. INRIA, jan. 2005. url : http://gallium.inria.fr/~fpottier/publis/simonet-pottier-hmg.ps.gz (cf. page 6).

[S96] Michael Sipser. *Introduction to the Theory of Computation*. 1st. International Thomson Publishing, 1996. isbn : 053494728X (cf. pages 5, 23).

[SP02] Christian Skalka et François Pottier. « Syntactic Type Soundness for HM(X) ». In : t. 75. TIP '02 (Workshop on Types in Programming). Dagstuhl, Germany, juil. 2002. url : http://gallium.inria.fr/~fpottier/publis/skalka-fpottier-tip-02.ps.gz (cf. page 93).

[TS96] Valery Trifonov et Scott Smith. « Subtyping Constrained Types ». In : SAS '06 (Static Analysis Symposium) 1145 (1996), p. 349–365 (cf. page 29).

[W96] Joe B. Wells. « Typability and Type Checking in the Second-Order lambda-Calculus Are Equivalent and Undecidable ». In : LICS '96 (Logic in Computer Science) (1996), p. 176–185 (cf. page 122).

[W95] Andrew Wright. « Simple Imperative Polymorphism ». In : LISP and Symbolic Computation (1995), p. 343–356 (cf. page 82).

[WF92] Andrew K. Wright et Matthias Felleisen. « A Syntactic Approach to Type Soundness ». In : Information & Computation 115 (1992), p. 38–94 (cf. pages 18, 57).

Index des termes

Index des règles d'inférence

Table des matières

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Sous-typage par saturation de contraintes, théorie et implémentation

Mots clés : Sous-typage, méthodes formelles, analyse statique, langages de programmation

Résumé : Cette thèse porte sur l'analyse statique de code par typage dans le but de détecter les erreurs dans les programmes avant leur exécution. Plus précisément, nous nous intéressons ici au domaine du sous-typage, dans lequel les propriétés du code sont représentées par des ensembles de contraintes de la forme $(\tau_1 \leqslant \tau_2)$. Nos mécanismes de vérification sont basés sur l'agrégation de contraintes de sous-typage et la vérification de leur compatibilité par saturation.

Le langage de base sur lequel nous travaillons est un ML étendu, muni de variants et d'un mécanisme de filtrage de motifs.

Après avoir défini un système de types de base pour notre langage, nous en présentons trois extensions originales : une amélioration du typage du filtrage de motifs basée en particulier sur l'ajout d'un opérateur de disjonction entre les contraintes de sous-typage, une alternative au mécanisme classique de généralisation permettant de distinguer les contraintes associées aux différents usages des paramètres des fonctions, et une formalisation des `GADT` basée sur une implémentation originale des types existentiels.

Bien qu'il soit possible de dériver directement une implémentation de ces systèmes, ce qui est principalement utile pour leur compréhension et leur prototypage, les performances des typeurs obtenus de la sorte ne sont pas suffisantes pour analyser des programmes de taille réelle. Nous présentons alors différentes techniques permettant à nos analyses de passer à l'échelle en pratique.

Title : Subtyping by Constraint Saturation, Theory and Implementation

Keywords : Subtyping, formal methods, static analysis, programming languages

Abstract : This PhD thesis focuses on static analysis of programs by type inference in order to detect program errors before their execution. More precisely, we focus here in the field of subtyping, where program properties are described by sets of constraints of the form $(\tau_1 \leqslant \tau_2)$. Our verification mechanisms are based on the aggregation of subtyping constraints and checking of their compatibility by saturation.

The base language on which we define our type systems is an ML-like language endowed with variants and pattern matching.

After the definition of a base type system for our language, we present three novel extensions : an improvement of type inference for the pattern matching based on the addition of the "or" operator between subtyping constraints, a new implementation of the generalization mechanism, and a formalization of `GADT` based on an novel implementation of existential types.

Despite the fact that it is possible to derive directly an implementation of our type systems from their rules, which is principally interesting for their comprehension and prototyping, the effectiveness of such typing methods are insufficient to analyze real-world programs. We then present several techniques to support the scalability of our method.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France