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 :

$$\varnothing, \varnothing \vdash \varnothing \vee \ \mathtt{let\ f = \lambda\ x\ .\ match\ x\ with\ I \to 42 \parallel\ S \to "..."\ in\ f\ I + 1}\ :\ \alpha \rhd \Phi$$

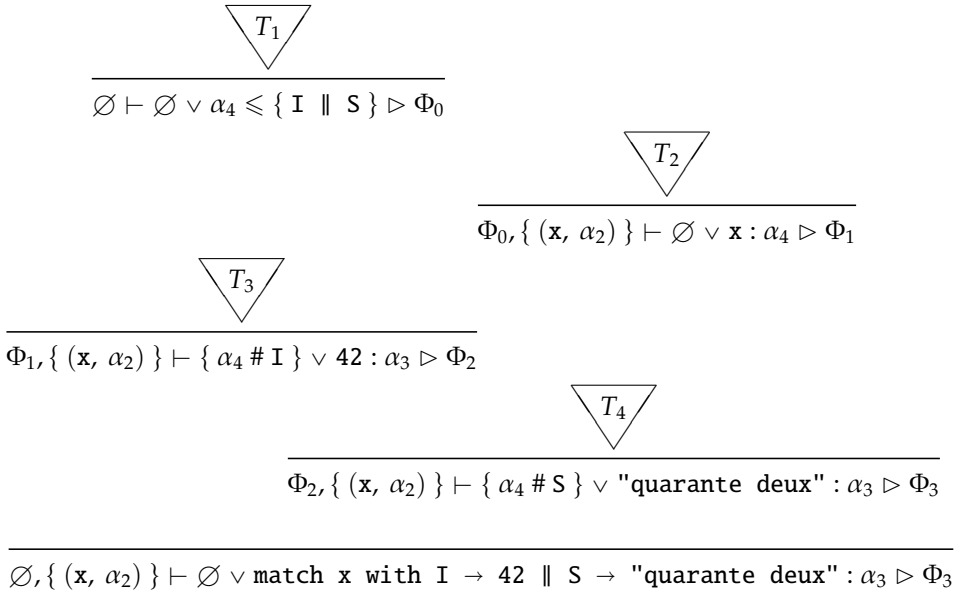
est néanmoins assez gros. Nous ne déroulerons ici que les sous-arbres intéressants dans le contexte de ce nouveau système de types.

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 :

$$\alpha_2 \# \mathtt{S} \vee \mathtt{string} \leqslant \alpha_3$$