Recherche de fonctions
par
unification modulo isomorphismes de types

Clément ALLAIN
Stage de M1 encadré par Gabriel RADANNE

Juillet 2021

Table des matières

1 Introduction **2**

2 Unification modulo isomorphismes de types **4**
2.1 Définitions préliminaires 4
2.1.1 Types 5
2.1.2 Théorie équationnelle 6
2.1.3 Matching et unification 8
2.2 Isomorphismes de types 9
2.2.1 Isomorphismes de types en λ-calcul 9
2.2.2 Isomorphismes de types en OCaml 10
2.3 Implémentation de l'unification modulo isomorphismes 10
2.3.1 Algorithme d'unification syntaxique 10
2.3.2 Algorithme d'unification modulo isomorphismes 12
2.3.3 Normalisation des types 13

3 Indexation **19**
3.1 Métriques de types 21
3.1.1 Nombre de variables uniques 21
3.1.2 Genre de la tête 22
3.1.3 Nombre de variables à la base 24
3.2 Critères d'unifiabilité 25
3.2.1 Premier critère 25
3.2.2 Deuxième critère 27
3.3 Utilisation d'un trie 31

4 `dowsindex` **33**

5 Conclusion **35**

Annexe **36**
5.1 Normalisation 36
5.1.1 Bonne formation d'une normalisée 36
5.1.2 Préservation de la bonne formation par application d'une substitution bien formée 37
5.1.3 Equivalence des unifiabilités 38
5.2 Conditions nécessaires d'unifiabilité 41
5.2.1 Première condition nécessaire 41
5.2.2 Deuxième condition nécessaire 42

Chapitre 1

Introduction

S'il est une chose dont on a besoin en programmation fonctionnelle, c'est bien de fonctions. Seulement voilà, ce n'est pas si simple. Mettre la main sur l'une d'entre elles en présence d'un environnement de grande taille s'avère parfois difficile, à tout le moins fastidieux. Aussi, remédions.

Il s'agit de mettre sur pied un système de recherche de fonctions pour le langage OCaml [19]. L'écosystème en est principalement composé des paquets OPAM — des centaines de milliers d'identificateurs. Un tel outil épargnerait à l'utilisateur la peine de fouiller telle ou telle bibliothèque dans l'espoir d'y trouver une fonction s'accordant à son usage.

Mais que chercher et comment ? Des travaux similaires ont été menés notamment par Rittri pour Lazy ML [8, 12] et Delahaye pour Coq [17]. Nous empruntons à notre tour à Rittri [8] le tableau 1.1.

Langage	Nom	Type
LCF ML [2]	`itlist`	$(\alpha \to \beta \to \beta) \to list(\alpha) \to \beta \to \beta$
Caml Light [15]	`list_it`	$(\alpha \to \beta \to \beta) \to list(\alpha) \to \beta \to \beta$
Haskell [20]	`foldr`	$(\alpha \to \beta \to \alpha) \to \alpha \to list(\beta) \to \alpha$
SML of New Jersey [6]	`fold`	$(\alpha \times \beta \to \beta) \to list(\alpha) \to \beta \to \beta$
Edinburgh SML [7]	`fold_right`	$(\alpha \times \beta \to \beta) \to \beta \to list(\alpha) \to \beta$

Table 1.1 – Variations sur `List.fold_left` d'OCaml dans plusieurs dialectes de Core-ML [3].

Plusieurs observations en émanent. Une recherche par identificateur en l'espèce — une fonction se comportant comme `List.fold_left` en OCaml — se heurterait à la diversité des noms potentiels. De même, une recherche syntaxique sur les types s'accommoderait mal des variations en un sens équivalentes de la forme du type. Pourtant, toutes ces fonctions font essentiellement le même travail ; cette diversité est due à l'arbitraire de l'auteur de la fonction.

Que faire, alors ? Une recherche par spécification s'avérerait bien plus satisfaisante, mais pas forcément décidable. On peut retenir comme spécification grossière le type d'une fonction — « *type as search key* » — tout en l'enrichissant par une approche sémantique et non plus purement syntaxique. La *recherche modulo isomorphisme de type* s'est imposée dans la littérature. Elle consiste intuitivement à autoriser des réarrangements triviaux dans les types :

permutations des paramètres et curryfication. Cette notion reste néanmoins à définir formellement, et surtout à décider algorithmiquement. Nous y consacrerons le chapitre 2.

Comme souligné par Rittri [12], il est bienvenu d'autoriser l'instantiation des types dans la recherche. Si la chose est seulement permise aux types issus de l'écosystème, il s'agit de *matching* ; si le type demandé peut aussi être instancié, on parlera d'*unification*. Gabriel Radanne a implémenté un algorithme d'unification présenté dans la dernière partie du chapitre 2.

Le système résultant ne saurait passer à l'échelle sans un travail sur les fonctions de l'écosystème préalable à la recherche. Dans le chapitre 3, nous proposerons une technique d'indexation reposant sur des conditions nécessaires d'unifiabilité. L'apport de cette technique se révélera substantiel.

Enfin, nous décrirons dans le chapitre 4 le fruit de notre travail : le programme `dowsindex`.

Chapitre 2

Unification modulo isomorphismes de types

Nous éclairons dans ce chapitre la notion d'*équivalence entre types* et la façon dont nous l'exploitons. Il s'agit de formaliser les réarrangements permis par des *isomorphismes de types*. Intuitivement, deux types sont *isomorphes* s'ils admettent deux fonctions de conversion permettant de passer de l'un à l'autre. L'*unifiabilité modulo isomorphismes*, ajoute à cela la capacité d'instancier les types. Nous présentons également l'implémentation de Gabriel Radanne qui a servi de point de départ à ce stage.

2.1 Définitions préliminaires

Commençons par poser quelques définitions : *types*, *axiome équationnel*, *théorie équationnelle*, *unifiabilité*. Pour cela, on se donne un ensemble dénombrable $\mathscr{V} = \{\alpha, \beta, \gamma, \delta, \ldots\}$ de symboles de variables.

Le langage OCaml permet de définir de nouveaux types : types-sommes, enregistrements... Nous aurons donc besoin de *constructeurs de types*. Nous les formalisons par la notion de *signature*.

Définition 2.1.1 (signature)
Une signature est un ensemble fini $\mathscr{F}$ de symboles de constructeurs muni d'une fonction d'arité $|\cdot|_{\mathscr{F}}$ de $\mathscr{F}$ dans $\mathbb{N}$ et tel que $\mathscr{V} \cap \mathscr{F} = \emptyset$. Les constructeurs d'arité nulle sont appelés constantes.

Exemple 2.1.2
Une signature $(\mathscr{F}, |\cdot|_{\mathscr{F}})$ pour OCaml serait telle que :
— $int \in \mathscr{F}$ avec $|int|_{\mathscr{F}} = 0$;
— $float \in \mathscr{F}$ avec $|float|_{\mathscr{F}} = 0$;
— $bool \in \mathscr{F}$ avec $|bool|_{\mathscr{F}} = 0$;
— $list \in \mathscr{F}$ avec $|list|_{\mathscr{F}} = 1$;
— $array \in \mathscr{F}$ avec $|array|_{\mathscr{F}} = 1$;
— ...

Soit une signature $(\mathscr{F}, |\cdot|_{\mathscr{F}})$ avec $\mathscr{F} = \{f, \ldots\}$.

2.1.1 Types

Définissons alors les *types*. Un *type* est soit :
- une variable ;
- le type **unit** ;
- un type-produit ;
- une type-flèche ;
- construit avec un symbole de la signature.

Définition 2.1.1.1 (type)
L'ensemble des types, noté T, est défini inductivement par :

$$\frac{}{\mathscr{V} \subseteq \mathrm{T}} \qquad \frac{}{\mathbf{unit} \in \mathrm{T}}$$

$$\frac{\tau_1 \in \mathrm{T} \quad \tau_2 \in \mathrm{T}}{\tau_1 \times \tau_2 \in \mathrm{T}} \qquad \frac{\tau_1 \in \mathrm{T} \quad \tau_2 \in \mathrm{T}}{\tau_1 \to \tau_2 \in \mathrm{T}} \qquad \frac{f \in \mathscr{F} \quad \overline{\tau} \in \mathrm{T}^{|f|_{\mathscr{F}}}}{f(\overline{\tau}) \in \mathrm{T}}$$

On s'autorisera à omettre les parenthèses pour les constantes de $\mathscr{F}$. De façon usuelle, ajoutons que $\cdot \times \cdot$ et $\cdot \to \cdot$ sont associatifs à droite avec $\cdot \times \cdot$ de précédence plus élevée.

Exemple 2.1.1.2
Avec la signature OCaml de l'exemple précédent, on a les types :
- $\mathbf{unit} \to int \times float$;
- $int \to (int \to \alpha) \to array(\alpha)$;
- $(\alpha \to \beta \to \alpha) \to \alpha \to list(\beta) \to \alpha$.

Définition 2.1.1.3 (variables d'un type)
L'ensemble des variables d'un type τ, noté $\mathrm{vars}(\tau)$, est défini inductivement par :

$$\begin{aligned}
\mathrm{vars}(\alpha) &= \{\alpha\} \\
\mathrm{vars}(\mathbf{unit}) &= \{\} \\
\mathrm{vars}(\tau_1 \times \tau_2) &= \mathrm{vars}(\tau_1) \cup \mathrm{vars}(\tau_2) \\
\mathrm{vars}(\tau_1 \to \tau_2) &= \mathrm{vars}(\tau_1) \cup \mathrm{vars}(\tau_2) \\
\mathrm{vars}(f(\overline{\tau})) &= \bigcup_{i \in [\![1\,;|f|_{\mathscr{F}}]\!]} \mathrm{vars}(\overline{\tau}_i)
\end{aligned}$$

Dans la suite, il nous faudra *instancier* un type : il s'agit de substituer une ou plusieurs variables par des types correspondants. Un type sera vu comme *moins général* qu'un autre s'il est obtenu en substituant dans ce dernier.

Définition 2.1.1.4 (substitution de types)
Une substitution de types est une fonction de $\mathscr{V}$ dans T.
On note Σ l'ensemble des substitution de types.

Définition 2.1.1.5 (domaine d'une substitution de types)
Le domaine d'une substitution de types σ, noté $\mathrm{dom}(\sigma)$, est l'ensemble :

$$\{\alpha \in \mathscr{V} \mid \hat{\sigma}(\alpha) \neq \alpha\}$$

Lorsque le domaine d'une substitution σ est fini de cardinal n — disons $\mathrm{dom}(\sigma) = \{\alpha_1, \dots, \alpha_n\}$ —, on s'autorisera à la noter $\{\alpha_1 \mapsto \sigma(\alpha_1), \dots, \alpha_n \mapsto \sigma(\alpha_n)\}$.

Définition 2.1.1.6 (extension d'une substitution de types)
L'extension d'une substitution de types σ, notée $\hat{\sigma}$, est l'unique endomorphisme de T dont la restriction à $\mathscr{V}$ est σ.
Autrement dit, $\hat{\sigma}$ est définie inductivement par :

$$\begin{aligned}
\hat{\sigma}(\alpha) &= \sigma(\alpha) \\
\hat{\sigma}(\mathbf{unit}) &= \mathbf{unit} \\
\hat{\sigma}(\tau_1 \times \tau_2) &= \hat{\sigma}(\tau_1) \times \hat{\sigma}(\tau_2) \\
\hat{\sigma}(\tau_1 \to \tau_2) &= \hat{\sigma}(\tau_1) \to \hat{\sigma}(\tau_2) \\
\hat{\sigma}(f(\tau_1, \dots, \tau_{|f|_{\mathscr{F}}})) &= f(\hat{\sigma}(\tau_1), \dots, \hat{\sigma}(\tau_{|f|_{\mathscr{F}}}))
\end{aligned}$$

Définition 2.1.1.7 ($\leqslant_\Sigma$)
Une substitution σ_1 est plus générale qu'une substitution σ_2, noté $\sigma_1 \leqslant_\Sigma \sigma_2$, *ssi* il existe une substitution σ'_1 telle que $\sigma_2 = \sigma'_1 \circ \sigma_1$.

Définition 2.1.1.8 (instance de type)
Un type τ_1 est une instance d'un type τ_2, noté $\tau_2 \leqslant_{\mathrm{T}} \tau_1$, *ssi* il existe une substitution σ telle que $\tau_1 = \hat{\sigma}(\tau_2)$.

Exemple 2.1.1.9
Toujours avec la signature OCaml, on a les instances :
— $\mathit{int} \times \mathit{int} \leqslant_{\mathrm{T}} \mathit{int} \times \mathit{int}$;
— $\alpha \to \beta \leqslant_T \beta \to \alpha \leqslant_{\mathrm{T}} \alpha \to \beta$;
— $\alpha \to \mathit{list}(\mathit{alpha}) \to \mathbf{unit} \leqslant_{\mathrm{T}} \mathit{int} \to \mathit{list}(\mathit{int}) \to \mathbf{unit}$.

2.1.2 Théorie équationnelle

Passées ces premières définitions, nous savons de qui nous parlons. Abordons maintenant la notion de *théorie équationnelle*. Le plus important est la donnée des *axiomes équationnels*, qui exprimeront les *isomorphismes de types* qui nous intéressent et feront toute la force et la difficulté de l'*unification modulo isomorphismes*. En découle naturellement une *théorie équationnelle* : la plus petite congruence contenant les instances des axiomes.

Définition 2.1.2.1 (axiome équationnel)
Un axiome équationnel est un couple de types de la forme $\tau_1 \cong_{\mathrm{T}} \tau_2$.

Définition 2.1.2.2 (système équationnel)
Un système équationnel est un ensemble d'axiomes équationnels.

Définition 2.1.2.3 (instance d'axiome équationnel)
Une instance d'un axiome équationnel $\tau_1 \cong_{\mathrm{T}} \tau_2$ est un couple de types (τ'_1, τ'_2) tel que $\tau_1 \leqslant_{\mathrm{T}} \tau'_1$ et $\tau_2 \leqslant_{\mathrm{T}} \tau'_2$.

Définition 2.1.2.4 (théorie équationnelle, $\mathscr{E}$-équivalence)
Soit un système équationnel $\mathscr{E}$.
La théorie équationnelle induite par $\mathscr{E}$, notée $\cdot \equiv^{\mathscr{E}}_{\mathrm{T}} \cdot$, est la plus petite congruence

par rapport à $\mathscr{F}$ sur les types contenant toutes les instances des axiomes équationnels de $\mathscr{E}$.
Autrement dit, c'est la plus petite relation binaire satisfaisant les règles d'inférence :

$$\frac{\tau_1 \cong_{\mathrm{T}} \tau_2 \in \mathscr{E}}{\hat{\sigma}(\tau_1) \equiv^{\mathscr{E}}_{\mathrm{T}} \hat{\sigma}(\tau_2)}\ (\equiv^{\mathscr{E}}_{\mathrm{T}}\text{-ax}) \qquad \frac{}{\tau \equiv^{\mathscr{E}}_{\mathrm{T}} \tau}\ (\equiv^{\mathscr{E}}_{\mathrm{T}}\text{-refl})$$

$$\frac{\tau_1 \equiv^{\mathscr{E}}_{\mathrm{T}} \tau_2 \qquad \tau_2 \equiv^{\mathscr{E}}_{\mathrm{T}} \tau_3}{\tau_1 \equiv^{\mathscr{E}}_{\mathrm{T}} \tau_3}\ (\equiv^{\mathscr{E}}_{\mathrm{T}}\text{-trans}) \qquad \frac{\tau_1 \equiv^{\mathscr{E}}_{\mathrm{T}} \tau_2}{\tau_2 \equiv^{\mathscr{E}}_{\mathrm{T}} \tau_1}\ (\equiv^{\mathscr{E}}_{\mathrm{T}}\text{-sym})$$

$$\frac{\tau_1 \equiv^{\mathscr{E}}_{\mathrm{T}} \tau_1'}{\tau_1 \times \tau_2 \equiv^{\mathscr{E}}_{\mathrm{T}} \tau_1' \times \tau_2}\ (\equiv^{\mathscr{E}}_{\mathrm{T}}\text{-}\times_1) \qquad \frac{\tau_2 \equiv^{\mathscr{E}}_{\mathrm{T}} \tau_2'}{\tau_1 \times \tau_2 \equiv^{\mathscr{E}}_{\mathrm{T}} \tau_1 \times \tau_2'}\ (\equiv^{\mathscr{E}}_{\mathrm{T}}\text{-}\times_2)$$

$$\frac{\tau_1 \equiv^{\mathscr{E}}_{\mathrm{T}} \tau_1'}{\tau_1 \to \tau_2 \equiv^{\mathscr{E}}_{\mathrm{T}} \tau_1' \to \tau_2}\ (\equiv^{\mathscr{E}}_{\mathrm{T}}\text{-}\to_1) \qquad \frac{\tau_2 \equiv^{\mathscr{E}}_{\mathrm{T}} \tau_2'}{\tau_1 \to \tau_2 \equiv^{\mathscr{E}}_{\mathrm{T}} \tau_1 \to \tau_2'}\ (\equiv^{\mathscr{E}}_{\mathrm{T}}\text{-}\to_2)$$

$$\frac{f \in \mathscr{F} \qquad i \in [\![1\,;|f|_{\mathscr{F}}]\!] \qquad \tau_i \equiv^{\mathscr{E}}_{\mathrm{T}} \tau_i'}{f(\tau_1,\ldots,\tau_i,\ldots,\tau_{|f|_{\mathscr{F}}}) \equiv^{\mathscr{E}}_{\mathrm{T}} f(\tau_1,\ldots,\tau_i',\ldots,\tau_{|f|_{\mathscr{F}}})}\ (\equiv^{\mathscr{E}}_{\mathrm{T}}\text{-}\mathscr{F})$$

Si $\tau_1 \equiv^{\mathscr{E}}_{\mathrm{T}} \tau_2$, on dit que τ_1 et τ_2 sont $\mathscr{E}$-équivalents.

Exemple 2.1.2.5
Prenons comme axiomes $\mathscr{E}$ la commutativité et l'associativité du produit :

$$\alpha \times \beta \cong_{\mathrm{T}} \beta \times \alpha \qquad (\times\text{-comm})$$
$$\alpha \times (\beta \times \gamma) \cong_{\mathrm{T}} (\alpha \times \beta) \times \gamma \qquad (\times\text{-assoc})$$

En OCaml, les équivalences suivantes sont alors correctes :
— $\alpha \times \mathit{int} \to \beta \equiv^{\mathscr{E}}_{\mathrm{T}} \mathit{int} \times \alpha \to \beta$;
— $\mathit{int} \times \mathit{float} \times \mathit{bool} \times \mathbf{unit} \equiv^{\mathscr{E}}_{\mathrm{T}} \mathit{float} \times \mathbf{unit} \times \mathit{int} \times \mathit{bool}$.
En revanche, celles-ci ne le sont pas :
— $\alpha \times \beta \to \gamma \equiv^{\mathscr{E}}_{\mathrm{T}} \alpha \to \beta \to \gamma$;
— $\alpha \to (\beta \times \gamma) \equiv^{\mathscr{E}}_{\mathrm{T}} (\alpha \to \beta) \times (\alpha \to \gamma)$.

Bien que la définition d'une théorie équationnelle ne mentionne pas la stabilité par substitution, c'en est un corollaire :

Lemme 2.1.2.6
Si deux types τ_1 et τ_2 sont $\mathscr{E}$-équivalents, alors pour toute substitution σ, $\hat{\sigma}(\tau_1)$ et $\hat{\sigma}(\tau_2)$ sont $\mathscr{E}$-équivalents.

Démonstration.
La preuve se fait par induction structurelle sur la dérivation de $\tau_1 \equiv^{\mathscr{E}}_{\mathrm{T}} \tau_2$ et par cas sur la dernière règle utilisée. Dans le cas ($\equiv^{\mathscr{E}}_{\mathrm{T}}$-ax), il s'agit de montrer que $\hat{\sigma}(\hat{\sigma}'(\tau_1))$ et $\hat{\sigma}(\hat{\sigma}'(\tau_2))$ sont $\mathscr{E}$-équivalents, où $\tau_1 \cong_{\mathrm{T}} \tau_2$ est un axiome de $\mathscr{E}$. Il suffit de remarquer que $\hat{\sigma} \circ \hat{\sigma}'$ est l'extension de la substitution $\hat{\sigma} \circ \sigma'$. Les autres cas sont triviaux. □

2.1.3 Matching et unification

Nous en arrivons enfin aux notions fondamentales de *matching* et *unification*. Un type sujet *matche* un type motif si l'on peut les rendre équivalents en instanciant le motif. Deux types sont *unifiables* si l'on peut les rendre équivalents en les instanciant simultanément. En fait, on peut réduire le *matching* à l'*unification* en substituant chaque variable du sujet par une constante n'apparaissant pas dans le motif (de manière exclusive).

Soit un système équationnel $\mathscr{E}$.

Définition 2.1.3.1 ($\mathscr{E}$-matcheur)
Une substitution σ est un $\mathscr{E}$-matcheur — ou encore matcheur modulo $\mathscr{E}$ — d'un type τ_1 pour un type τ_2 *ssi* :

$$\tau_1 \equiv_{\mathrm{T}}^{\mathscr{E}} \hat{\sigma}(\tau_2)$$

Définition 2.1.3.2 ($\mathscr{E}$-matchabilité)
Un type τ_1 est $\mathscr{E}$-matchable — ou encore matchable modulo $\mathscr{E}$ — avec un type τ_2 *ssi* τ_1 admet un $\mathscr{E}$-matcheur pour τ_2.

Définition 2.1.3.3 (problème d'$\mathscr{E}$-matchabilité)
Le problème d'$\mathscr{E}$-matchabilité — ou encore problème de matchabilité modulo $\mathscr{E}$ — consiste à déterminer si un type est matchable avec un autre.

Définition 2.1.3.4 (problème d'$\mathscr{E}$-matching)
Soient τ_1 et τ_2 deux types.
Le problème d'$\mathscr{E}$-matching — ou encore problème de matching modulo $\mathscr{E}$ — consiste à trouver un $\mathscr{E}$-matcheur de τ_1 pour τ_2.

Exemple 2.1.3.5
Reprenons le système équationnel $\mathscr{E}$ de l'exemple précédent. Les types OCaml suivants sont $\mathscr{E}$-matchables :

— $int \times \alpha \times float$ et $\beta \times \alpha$ avec $\{\beta \mapsto int \times float\}$;

— $int \times float \to int$ et $\alpha \times \beta \to \beta$ avec $\{\alpha \mapsto float, \beta \mapsto int\}$.

Définition 2.1.3.6 ($\mathscr{E}$-unificateur)
Une substitution σ est un $\mathscr{E}$-unificateur des types τ_1 et τ_2 *ssi* :

$$\hat{\sigma}(\tau_1) \equiv_{\mathrm{T}}^{\mathscr{E}} \hat{\sigma}(\tau_2)$$

Définition 2.1.3.7 ($\mathscr{E}$-unifiabilité)
Deux types τ_1 et τ_2 sont $\mathscr{E}$-unifiables — ou encore unifiables modulo $\mathscr{E}$ —, noté $\tau_1 \sim_{\mathrm{T}}^{\mathscr{E}} \tau_2$, *ssi* ils admettent un $\mathscr{E}$-unificateur.

Définition 2.1.3.8 (problème d'$\mathscr{E}$-unifiabilité)
Le problème d'$\mathscr{E}$-unifiabilité — ou encore problème d'unifiabilité modulo $\mathscr{E}$ — consiste à déterminer si deux types sont $\mathscr{E}$-unifiables.

Définition 2.1.3.9 (problème d'$\mathscr{E}$-unification)
Soient τ_1 et τ_2 deux types.
Le problème d'$\mathscr{E}$-unification — ou encore problème d'unification modulo $\mathscr{E}$ — consiste à trouver un $\mathscr{E}$-unificateur de τ_1 et τ_2.

Lorsque $\mathscr{E}$ est vide, nous parlerons d'unification syntaxique ; dans le cas contraire, d'unification sémantique.

Exemple 2.1.3.10
Avec le même système $\mathscr{E}$, les types OCaml suivants sont $\mathscr{E}$-unifiables :

— $int \times \alpha$ et $\beta \times float$ avec $\{\alpha \mapsto float, \beta \mapsto int\}$;

— $\alpha \times \beta \to list(\alpha)$ et $list(\gamma) \times int \times float \to \gamma$
avec $\{\alpha \mapsto float \times int, \beta \mapsto list(\gamma), \gamma \mapsto list(\alpha)\}$.

2.2 Isomorphismes de types

Ainsi parés, nous sommes à présent en mesure d'aborder le cœur de ce chapitre. Nous allons définir la notion d'équivalence entre types de notre système de recherche.

Les travaux de Mikael Rittri [8, 12] sur lesquels nous nous appuyons ont assis celle d'*équivalence modulo isomorphismes de types*. La chose a fait l'objet de plusieurs recherches importantes, menées notamment par Giusseppe Longo, Kim Bruce, Roberto Di Cosmo [9, 10, 11, 14], Sergei Soloviev [5, 13], Paliath Narendran, Frank Pfenning et Richard Statman [16].

2.2.1 Isomorphismes de types en λ-calcul

Dans cette section, on ignore les constructeurs de types : $\mathscr{F}$ est l'ensemble vide.

Les travaux cités auparavant décrivent les *isomorphismes de types* dans plusieurs versions du λ-calcul simplement typé. Notons Λ^1 le λ-calcul simplement typé avec paires et élément terminal et $=_{\Lambda^1}$ l'égalité associée (pour plus de détails, voir par exemple [14]). La notion d'*isomorphisme de types* dans Λ^1 se définit ainsi :

Définition 2.2.1.1 (isomorphisme de types dans Λ^1)
Deux types τ_1 et τ_2 sont isomorphes dans Λ^1 *ssi* il existe deux λ-termes $f : \tau_1 \to \tau_2$ et $g : \tau_2 \to \tau_1$ tels que $f \circ g =_{\Lambda^1} \mathrm{id}_{\tau_2}$ et $g \circ f =_{\Lambda^1} \mathrm{id}_{\tau_1}$.

On peut donner une caractérisation équationnelle des isomorphismes de Λ^1. Ils correspondent aux isomorphismes valides dans les *catégories cartésiennes fermées*. Longo, Bruce, Di Cosmo [9] et Soloviev [5] ont démontré par deux méthodes différentes que les axiomes équationnels ci-dessous sont corrects et complets. On notera $\mathscr{E}^1$ la théorie équationnelle induite.

$$\begin{aligned}
\alpha \times \beta &\cong_{\mathrm{T}} \beta \times \alpha && (\times\text{-comm})\\
\alpha \times (\beta \times \gamma) &\cong_{\mathrm{T}} (\alpha \times \beta) \times \gamma && (\times\text{-assoc})\\
\mathbf{unit} \times \alpha &\cong_{\mathrm{T}} \alpha && (\times\text{-}\mathbf{unit})\\
(\alpha \times \beta) \to \gamma &\cong_{\mathrm{T}} \alpha \to (\beta \to \gamma) && (\text{curry})\\
\mathbf{unit} \to \alpha &\cong_{\mathrm{T}} \alpha && (\text{curry-}\mathbf{unit})\\
\alpha \to (\beta \times \gamma) &\cong_{\mathrm{T}} (\alpha \to \beta) \times (\alpha \to \gamma) && (\text{dist})\\
\alpha \to \mathbf{unit} &\cong_{\mathrm{T}} \mathbf{unit} && (\text{dist-}\mathbf{unit})
\end{aligned}$$

Néanmoins, Narendran, Pfenning et Statman [16] ont établi que, si le *matching* modulo $\mathscr{E}^1$ est NP-complet, l'unification modulo $\mathscr{E}^1$ est indécidable. C'est ce qui a poussé Rittri [12] à se concentrer sur les cinq premiers axiomes correspondant aux *isomorphismes linéaires* (pour plus de détails, voir [12]) complets

dans les *catégories monoïdales fermées symétriques* [13]. L'unification modulo cette nouvelle théorie est NP-complète [16]. Cet abandon des deux axiomes de distributivité (dist) et (dist-**unit**) — et donc de la complétude dans Λ^1 — s'avère d'après lui bénin, arguant que ces isomorphismes sont peu utiles en pratique.

2.2.2 Isomorphismes de types en OCaml

En outre, Di Cosmo [10] a étudié les isomorphismes de types en Core-ML. De même, il en a tiré une axiomatisation équationnelle complète comportant de nouveaux isomorphismes. L'équivalence est alors décidable par un algorithme qu'il introduit dans [14]. Malheureusement, cet algorithme ne s'adapte a priori pas directement au matching ou l'unification.

L'implémentation suit les préconisations de Rittri en renonçant à l'axiome (curry-**unit**). En effet, bien qu'utile en pratique pour simuler une expression paresseuse, il complexifie l'algorithme d'unification. Ne s'agissant pas de l'isomorphisme le plus important, cela affecte peu la qualité de la recherche.

Reste donc les quatre premiers isomorphismes : ($\times$-comm), ($\times$-assoc), ($\times$-**unit**), (curry). Ils capturent exactement les réarrangements qui nous intéressent, en particulier la permutation des paramètres. Relevons toutefois que, si ces isomorphismes sont corrects pour Core-ML [11], leur correction pour OCaml n'est pas démontrée. Nous admettons ici qu'ils se « propagent » aux constructeurs de types, c'est-à-dire la congruence sur la signature OCaml.

Dans le reste de ce rapport, nous traiterons des types OCaml avec constructeurs. Nous dénoterons par $\mathscr{E}$ le système équationnel formé par les quatre axiomes, $\cdot \equiv_{\mathrm{T}} \cdot$ la théorie équationnelle induite par $\mathscr{E}$ et $\cdot \sim_{\mathrm{T}} \cdot$ l'$\mathscr{E}$-unifiabilité. Nous parlerons d'équivalence pour l'$\mathscr{E}$-équivalence, de matching pour l'$\mathscr{E}$-matching, d'unifiabilité pour l'$\mathscr{E}$-unifiabilité et d'unification pour l'$\mathscr{E}$-unification.

Exemple 2.2.2.1
Les types suivants sont unifiables :

— $int \to float \to int \sim_{\mathrm{T}} float \to \alpha$ avec $\{\alpha \mapsto int \to int\}$;

— $int \to \alpha \to \mathbf{unit} \sim_{\mathrm{T}} int \to \mathbf{unit}$ avec $\{\alpha \mapsto \mathbf{unit}\}$;

— $int \to \alpha \to \mathbf{unit} \sim_{\mathrm{T}} int \to int \to int \to \mathbf{unit}$ avec $\{\alpha \mapsto int \times int\}$;

— $(\alpha \to \beta \to \alpha) \to \alpha \to list(\beta) \to \alpha \sim_{\mathrm{T}} list(\gamma) \times \delta \to (\gamma \times \delta \to \delta) \to \delta$ avec $\{\gamma \mapsto \beta, \delta \mapsto \alpha\}$.

2.3 Implémentation de l'unification modulo isomorphismes

2.3.1 Algorithme d'unification syntaxique

Avant d'aborder l'implémentation, revenons sur l'ancêtre commun des algorithmes d'unification sémantique, dû à Herbrand [1], Martelli et Montanari [4]. Leur algorithme d'unification syntaxique consiste à transformer le problème initial par applications successives d'un certain nombre de règles afin d'aboutir à une forme résolue — essentiellement une substitution. La conception d'un algorithme d'unification sémantique repart souvent de ce principe, en modifiant les règles ou en en ajoutant.

Définissons tout d'abord les objets manipulés par l'algorithme : *équations de types* et *problèmes équationnels*.

Définition 2.3.1.1 (équation de types)
Une équation de types est un couple de types de la forme $\tau_1 \stackrel{?}{=} \tau_2$.

Définition 2.3.1.2 (équation de types en forme résolue)
Une équation de types est en forme résolue *ssi* elle est de la forme $\alpha \stackrel{?}{=} \tau$.

Définition 2.3.1.3 (problème équationnel)
Un problème équationnel est un ensemble fini d'équations de types.

Définition 2.3.1.4 (problème en forme résolue)
Un problème est en forme résolue *ssi* il est de la forme :

$$\{\alpha_1 \stackrel{?}{=} \tau_1, \ldots, \alpha_n \stackrel{?}{=} \tau_n\}$$

où chaque α_i n'apparaît qu'une seule fois. Il induit alors la substitution :

$$\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}$$

On étend naturellement substitution, ensemble de variables et unificateur aux équations de types et problèmes équationnels.

Ceci étant fait, voyons maintenant comment *transformer* un problème équationnel. Dans cette présentation, six règles suffisent :

- (effacer) efface une équation triviale ;
- (orienter) réoriente une équation de la forme $\tau \stackrel{?}{=} \alpha$, rendant (remplacer) potentiellement applicable ;
- (décomposer-$\times$), (décomposer-$\rightarrow$) et (décomposer-$\mathscr{F}$) décomposent respectivement un produit, une flèche et un type formé à l'aide d'un constructeur ;
- (substituer) substitue à l'aide d'une équation en forme résolue.

Définition 2.3.1.5 (transformation d'un problème équationnel)
Un problème équationnel P_1 se transforme en un problème P_2, noté $P_1 \rightsquigarrow P_2$, à l'aide les règles suivantes :

$$P \cup \{\tau \stackrel{?}{=} \tau\} \rightsquigarrow P \text{ (effacer)}$$

$$\frac{\tau \notin \mathscr{V}}{P \cup \{\tau \stackrel{?}{=} \alpha\} \rightsquigarrow P \cup \{\alpha \stackrel{?}{=} \tau\}} \text{ (orienter)}$$

$$P \cup \{\tau_1 \times \tau_2 \stackrel{?}{=} \tau'_1 \times \tau'_2\} \rightsquigarrow P \cup \{\tau_1 \stackrel{?}{=} \tau'_1, \tau_2 \stackrel{?}{=} \tau'_2\} \text{ (décomposer-}\times)$$

$$P \cup \{\tau_1 \rightarrow \tau_2 \stackrel{?}{=} \tau'_1 \rightarrow \tau'_2\} \rightsquigarrow P \cup \{\tau_1 \stackrel{?}{=} \tau'_1, \tau_2 \stackrel{?}{=} \tau'_2\} \text{ (décomposer-}\rightarrow)$$

$$\begin{array}{c} P \cup \{f(\tau_1, \ldots, \tau_{|f|_{\mathscr{F}}}) \stackrel{?}{=} f(\tau'_1, \ldots, \tau'_{|f|_{\mathscr{F}}})\} \rightsquigarrow \\ P \cup \{\tau_1 \stackrel{?}{=} \tau'_1, \ldots, \tau_{|f|_{\mathscr{F}}} \stackrel{?}{=} \tau'_{|f|_{\mathscr{F}}}\} \end{array} \text{ (décomposer-}\mathscr{F})$$

$$\frac{\alpha \in \text{vars}(P) \qquad \alpha \notin \text{vars}(\tau)}{P \cup \{\alpha \stackrel{?}{=} \tau\} \rightsquigarrow \{\alpha \mapsto \tau\}(P) \cup \{\alpha \stackrel{?}{=} \tau\}} \text{ (substituer)}$$

Explicitons l'algorithme d'unification syntaxique résultant $\mathscr{A}$. Etant donné deux types τ_1 et τ_2 à unifier, $\mathscr{A}(\tau_1, \tau_2)$ applique ces transformations autant que possible en partant du problème $\{\tau_1 \stackrel{?}{=} \tau_2\}$. Il échoue si le problème obtenu n'est pas en forme résolue et retourne la substitution induite sinon.

Cet algorithme est correct et complet en ce sens : deux types τ_1 et τ_2 admettent un unificateur σ minimal pour $\leqslant_\Sigma$ *ssi* $\mathscr{A}(\tau_1, \tau_2)$ retourne σ. Nous ne le redémontrons pas ici.

Exemple 2.3.1.6
Appliquons $\mathscr{A}$ aux types $int \to list(\alpha) \to \beta$ et $\gamma \to \delta \to \gamma$.

$$\begin{aligned}
\{int \to list(\alpha) \to \beta \stackrel{?}{=} \gamma \to \delta \to \gamma\} &\leadsto && \text{(decomposer-}\to\text{)}\\
\{int \stackrel{?}{=} \gamma, list(\alpha) \to \beta \stackrel{?}{=} \delta \to \gamma\} &\leadsto && \text{(orienter)}\\
\{\gamma \stackrel{?}{=} int, list(\alpha) \to \beta \stackrel{?}{=} \delta \to \gamma\} &\leadsto && \text{(substituer)}\\
\{\gamma \stackrel{?}{=} int, list(\alpha) \to \beta \stackrel{?}{=} \delta \to int\} &\leadsto && \text{(décomposer-}\to\text{)}\\
\{\gamma \stackrel{?}{=} int, list(\alpha) \stackrel{?}{=} \delta, \beta \stackrel{?}{=} int\} &\leadsto && \text{(orienter)}\\
\{\gamma \stackrel{?}{=} int, \delta \stackrel{?}{=} list(\alpha), \beta \stackrel{?}{=} int\} & && \text{en forme résolue}
\end{aligned}$$

2.3.2 Algorithme d'unification modulo isomorphismes

Le matching et l'unification modulo $\mathscr{E}$ sont décidables [16]. Gabriel Radanne a implémenté un algorithme d'unification. Néanmoins, comme nous l'avons vu auparavant, le matching s'y réduit. L'usage indique d'ailleurs qu'il n'est généralement pas souhaitable d'instancier le type demandé ; c'est pourquoi nous en avons fait le comportement par défaut, avec la possibilité d'expliciter les variables de types instanciables.

L'algorithme d'unification modulo $\mathscr{E}$ est conçu à partir d'un algorithme d'unification modulo associativité et commutativité (AC) développé par Boudet [18]. Ce dernier possède les mêmes traits que l'algorithme d'unification syntaxique présenté précédemment : il s'agit de transformer un problème à l'aide d'un jeu de règles. Dans un souci d'optimisation, les objets qu'il manipule s'avèrent toutefois plus élaborés. Le plus important est l'ajout d'une règle de transformation résolvant un *problème AC élémentaire* n'impliquant que des termes simples (variables et constantes) et un unique opérateur AC.

L'algorithme de Boudet a été adapté de la manière suivante :
- la représentation des termes à base de pointeurs se traduit par d'autres mécanismes d'indirection en OCaml ;
- les types sont *normalisés* : les produits sont « aplatis » pour former des uplets compatibles avec la procédure d'unification AC élémentaire ;
- une règle supplémentaire permet de traiter l'unification des flèches que l'isomorphisme (curry) rend non triviale.

Mentionnons un dernier trait de l'implémentation de l'unification. La puissance des quatre isomorphismes réunis — en particulier ($\times$-**unit**) — permet de réduire le nombre de paramètres d'une fonction en instanciant une ou plusieurs variables de type en **unit**. L'utilisateur pourrait ainsi positionner une variable « joker » potentiellement annulable en paramètre : $int \to \alpha \to list(int)$. Néanmoins, cela présente l'inconvénient de grossir les résultats de manière superflue

lorsque ce n'est pas l'effet souhaité. Conformément à l'article de Rittri [12], l'instanciation d'une variable de type en **unit** est donc exclue. Cette objection n'est pas sans importance ; dans le chapitre qui suit, la correction du premier critère d'unifiabilité la nécessite. Formellement, il s'agit d'exiger de l'unificateur qu'il soit *bien formé* :

Définition 2.3.2.1 (substitution de types bien formée)
Une substitution de types σ est bien formée, noté σ **bf**, *ssi* :

$$\forall \alpha \in \mathscr{V},\ \sigma(\alpha) \not\equiv_{\mathrm{T}} \mathbf{unit}$$

On note $\Sigma^{\mathbf{bf}}$ l'ensemble des substitutions de types bien formées.

2.3.3 Normalisation des types

Attachons-nous à formaliser la *normalisation des types*. Pour ce faire, on définit d'abord les *types normalisés* — c'est-à-dire le résultat de la normalisation — puis la fonction de normalisation elle-même.

Etant donné un ensemble A, on note $A^{\#}$ l'ensemble des multi-ensembles finis sur A et $\cdot +_A^{\#} \cdot$, $|\cdot|_A^{\#}$ et $\cdot \subseteq_A^{\#} \cdot$ respectivement la somme, le cardinal et l'inclusion sur $A^{\#}$. Nous n'explicitons pas la représentation. On suppose toutefois qu'il est licite d'écrire $\mathrm{N}^{\#} \subseteq \mathrm{N}$ dans la définition des *types normalisés*. En Coq, on utiliserait typiquement des listes ou vecteurs.

Définissons donc les *types normalisés*. Un *type* normalisé est soit :
— une variable ;
— un uplet de types normalisés représenté par un multi-ensemble ;
— une flèche où les paramètres sont regroupés dans un uplet ;
— construit avec un symbole de la signature.

Définition 2.3.3.1 (type normalisé)
L'ensemble des types normalisés, noté N, est défini inductivement par :

$$\frac{}{\mathscr{V} \subseteq \mathrm{N}} \qquad \frac{}{\mathrm{N}^{\#} \subseteq \mathrm{N}}$$

$$\frac{\nu^{\#} \in \mathrm{N}^{\#} \qquad \nu \in \mathrm{N}}{\nu^{\#} \to \nu \in \mathrm{N}} \qquad \frac{f \in \mathscr{F} \qquad \overline{\nu} \in \mathrm{N}^{|f|_{\mathscr{F}}}}{f(\overline{\nu}) \in \mathrm{N}}$$

Exemple 2.3.3.2
On a les types normalisés :
— $\{\![int, \{\![]\!\}, \alpha]\!\}$;
— $\{\![]\!\} \to \{\![]\!\} \to \{\![]\!\}$;
— $\{\![list(\{\![\alpha, int]\!\})]\!\} \to \{\![\alpha, \beta]\!\}$.

Afin de désigner la forme d'un type normalisé de manière concise, introduisons par ailleurs la notion de *genre*.

Définition 2.3.3.3 (genre de types normalisés)
L'ensemble des genres de types normalisés, noté $\mathscr{G}$, est défini par :

$$\frac{}{\mathbf{var} \in \mathscr{G}} \qquad \frac{}{\mathbf{uplet} \in \mathscr{G}} \qquad \frac{}{\mathbf{fleche} \in \mathscr{G}} \qquad \frac{f \in \mathscr{F}}{\mathbf{cons}_f \in \mathscr{G}}$$

Définition 2.3.3.4 (genre d'un type normalisé)
Le genre d'un type normalisé ν, noté $[\nu]$, est défini par :

$$\begin{aligned}[\alpha] &= \mathbf{var} \\ [\{\![\nu_1, \ldots, \nu_n]\!\}] &= \mathbf{uplet} \\ [\nu^\# \to \nu] &= \mathbf{fleche} \\ [f(\overline{\nu})] &= \mathbf{cons}_f\end{aligned}$$

La normalisation que nous nous apprêtons à définir offre certaines garanties qui ne sont pas contenues dans la définition des types normalisés :
— les éléments d'un uplet ne sont pas eux-mêmes des uplets ;
— le membre droit d'une flèche n'est pas lui-même une flèche ;
— un uplet qui n'est pas le membre gauche d'une flèche n'est pas un simplet.
C'est pourquoi nous parlerons la plupart du temps de types normalisés *bien formés*. Les fonctions les manipulant se devront de préserver cette qualité.

Définition 2.3.3.5 (type normalisé bien formé)
Les prédicats sur les types normalisés « est bien formé », noté **bf**, et « est presque bien formé », noté **pbf**, sont définis inductivement :

$$\frac{}{\alpha\ \mathbf{bf}} \qquad \frac{\nu^\#\ \mathbf{pbf} \qquad |\nu^\#|^\#_{\mathrm{N}} \neq 1}{\nu^\#\ \mathbf{bf}}$$

$$\frac{\nu^\#\ \mathbf{pbf} \qquad \nu\ \mathbf{bf} \qquad [\nu] \neq \mathbf{fleche}}{\nu^\# \to \nu\ \mathbf{bf}} \qquad \frac{f \in \mathscr{F} \qquad \forall i \in [\![1\,;|f|_{\mathscr{F}}]\!],\ \nu_i\ \mathbf{bf}}{f(\nu_1, \ldots, \nu_{|f|_{\mathscr{F}}})\ \mathbf{bf}}$$

$$\frac{}{\alpha\ \mathbf{pbf}} \qquad \frac{\forall i \in [\![1\,;n]\!],\ \nu_i\ \mathbf{bf} \wedge [\nu_i] \neq \mathbf{tuple}}{\{\![\nu_1, \ldots, \nu_n]\!\}\ \mathbf{pbf}}$$

$$\frac{\nu^\#\ \mathbf{pbf} \qquad \nu\ \mathbf{bf} \qquad [\nu] \neq \mathbf{fleche}}{\nu^\# \to \nu\ \mathbf{pbf}} \qquad \frac{f \in \mathscr{F} \qquad \forall i \in [\![1\,;|f|_{\mathscr{F}}]\!],\ \nu_i\ \mathbf{bf}}{f(\nu_1, \ldots, \nu_{|f|_{\mathscr{F}}})\ \mathbf{pbf}}$$

On note $\mathrm{N}^{\mathbf{bf}}$ l'ensemble des types normalisés bien formés et $\mathrm{N}^{\mathbf{pbf}}$ l'ensemble des types presque bien formés.

Lemme 2.3.3.6
Si un type normalisé est bien formé, alors il est presque bien formé.

Démonstration.
Trivial avec la définition de **bf** et **pbf**. □

Exemple 2.3.3.7
Les types normalisés suivants sont bien formés :
— $\{\![int, list(\{\![\alpha, int]\!\})]\!\}$;
— $\{\![int]\!\} \to \{\![\{\![]\!\} \to \alpha, int]\!\}$.
Ceux-ci ne le sont pas :
— $\{\![int, \{\![]\!\}, \alpha]\!\}$;
— $\{\![]\!\} \to \{\![]\!\} \to \{\![]\!\}$;

— $\{\![\,]\!\} \to \{\![int]\!\}$.

Nous avons maintenant tout ce qu'il faut pour définir la *normalisation* des types. Elle consiste à tirer parti des quatre isomorphismes en :
— « aplatissant » les produits en vertu des axiomes ($\times$-comm) et ($\times$-assoc) ;
— éliminant les types **unit** en vertu de ($\times$-**unit**) ;
— regroupant les paramètres à gauche des flèches en vertu de (curry).

Définition 2.3.3.8 (normalisée d'un type)
La normalisée d'un type τ, notée $\mathrm{norm}(\tau)$, est définie inductivement par :

$$
\begin{array}{rcll}
\mathrm{norm}(\alpha) & = & \alpha & \\
\mathrm{norm}(\mathbf{unit}) & = & \{\![\,]\!\} & \\
\mathrm{norm}(\tau_1 \times \tau_2) & = & \mathrm{norm}''(\mathrm{norm}'(\mathrm{norm}(\tau_1)) +^{\#}_{\mathrm{N}} \mathrm{norm}'(\mathrm{norm}(\tau_2))) & \\
\mathrm{norm}(\tau_1 \to \tau_2) & = & (\mathrm{norm}'(\mathrm{norm}(\tau_1)) +^{\#}_{\mathrm{N}} \nu_2^{\#}) \to \nu_2 & \text{si } \mathrm{norm}(\tau_2) = \nu_2^{\#} \to \nu_2 \\
\mathrm{norm}(\tau_1 \to \tau_2) & = & \mathrm{norm}'(\mathrm{norm}(\tau_1)) \to \mathrm{norm}(\tau_2) & \text{sinon} \\
\mathrm{norm}(f(\tau_1, \ldots, \tau_{|f|_{\mathscr{F}}})) & = & f(\mathrm{norm}(\tau_1), \ldots, \mathrm{norm}(\tau_{|f|_{\mathscr{F}}})) & \\
\mathrm{norm}'(\nu) & = & \nu & \text{si } \nu \in \mathrm{N}^{\#} \\
\mathrm{norm}'(\nu) & = & \{\![\nu]\!\} & \text{sinon} \\
\mathrm{norm}''(\{\![\nu]\!\}) & = & \nu & \\
\mathrm{norm}''(\nu^{\#}) & = & \nu^{\#} &
\end{array}
$$

Exemple 2.3.3.9
Quelques exemples de normalisation :
— $\mathrm{norm}(int \times float \times bool \times int) = \{\![int, int, float, bool]\!\}$;
— $\mathrm{norm}(\mathbf{unit} \times int) = int$;
— $\mathrm{norm}(\mathbf{unit} \times int \times \alpha) = \{\![int, \alpha]\!\}$;
— $\mathrm{norm}(\mathbf{unit} \times \mathbf{unit} \times \mathbf{unit}) = \{\![\,]\!\}$;
— $\mathrm{norm}(\alpha \times int \to \beta \to \gamma) = \{\![\alpha, int, \beta]\!\} \to \gamma$.

Comme attendu, la normalisation donne des types normalisés bien formés :

Théorème 2.3.3.10
La normalisée de tout type est bien formée.

Démonstration.
Voir annexe. □

Nous savons comment passer des types aux types types normalisés. Reste à lier des deux domaines. Ce à quoi nous voulons arriver est un théorème d'équivalence pour l'unification. Il s'agit donc de définir la notion d'unifiabilité pour les types normalisés. De la même manière qu'auparavant, voyons tout d'abord par la notion de *substitution* :

Définition 2.3.3.11 (substitution de types normalisés)
Une substitution de types normalisés est une fonction de $\mathscr{V}$ dans N.
On note Θ l'ensemble des substitution de types normalisés.

On définit le domaine d'une substitution de types normalisés θ comme précédemment. Lorsque ce domaine est fini, on s'autorisera de même à la noter sous la forme $\{\alpha_1 \mapsto \theta(\alpha_1), \ldots, \alpha_n \mapsto \theta(\alpha_n)\}$, où $\{\alpha_1, \ldots, \alpha_n\} = \mathrm{dom}(\theta)$.

Définition 2.3.3.12 (extension d'une substitution de types normalisés)
L'extension d'une substitution de types normalisés θ, notée $\hat{\theta}$, est définie inductivement par :

$$
\begin{aligned}
\hat{\theta}(\alpha) &= \alpha \\
\hat{\theta}(\nu^{\#}) &= \tilde{\theta}(\nu^{\#}) \\
\hat{\theta}(\nu^{\#} \to \nu) &= (\tilde{\theta}(\nu^{\#}) +^{\#}_{\mathrm{N}} \nu^{\#\prime}) \to \nu' && \text{si } \hat{\theta}(\nu) = \nu^{\#\prime} \to \nu' \\
\hat{\theta}(\nu^{\#} \to \nu) &= \tilde{\theta}(\nu^{\#}) \to \hat{\theta}(\nu) && \text{sinon} \\
\hat{\theta}(f(\nu_1, \ldots, \nu_{|f|_{\mathscr{F}}})) &= f(\hat{\theta}(\nu_1), \ldots, \hat{\theta}(\nu_{|f|_{\mathscr{F}}})) \\
\tilde{\theta}(\{[\nu_1, \ldots, \nu_n]\}) &= \sum_{i=1}^{n}{}^{\#}_{\mathrm{N}} \, \mathrm{norm}'(\hat{\theta}(\nu_i))
\end{aligned}
$$

Exemple 2.3.3.13
Quelques exemples d'applications de substitutions :
— $\{\alpha \mapsto \{[int, float]\}, \beta \mapsto bool\}$ appliquée à $\{[\alpha, \beta, \gamma]\}$ donne $\{[int, float, bool, \gamma]\}$;
— $\{\alpha \mapsto \{[int]\} \to \{[]\}\}$ appliquée à $\{[\alpha]\} \to \alpha$ donne $\{[\{[int]\} \to \{[]\}, int]\} \to \{[]\}$;
— $\{\alpha \mapsto \{[int, int]\}\}$ appliquée à $\{[\alpha]\} \to \alpha$ donne $\{[int, int]\} \to \{[int, int]\}$.

On imposera également aux substitutions d'être *bien formées*, c'est-à-dire aux substitués d'être eux-mêmes bien formés et différents de la normalisée de **unit**.

Définition 2.3.3.14 (substitution de types normalisés bien formée)
Une substitution de types θ est bien formée, noté θ **bf**, *ssi* :

$$\forall \alpha \in \mathscr{V},\ \theta(\alpha)\ \mathbf{bf} \land \theta(\alpha) \neq \{[]\}$$

On note $\Theta^{\mathbf{bf}}$ l'ensemble des substitutions de types normalisés bien formées.

On montre que l'application d'une substitution bien formée préserve la bonne formation des types normalisés.

Théorème 2.3.3.15
Pour tout ν dans $\mathrm{N}^{\mathbf{bf}}$ et θ dans $\Theta^{\mathbf{bf}}$, $\hat{\theta}(\nu)$ est bien formé.

Démonstration.
Voir annexe. □

Le dernier ingrédient pour parler d'unification est une relation d'*équivalence* sur les types normalisés. Une première approche consiste à repartir de la définition de l'équivalence sur les types sans les axiomes et les règles pour le produit et en y ajoutant des règles pour les uplets :

Définition 2.3.3.16 (types normalisés équivalents)
La relation binaire sur les types normalisés $\cdot \equiv_{\mathrm{N}} \cdot$ est définie inductivement par :

$$\frac{}{\nu \equiv_{\mathrm{N}} \nu}\ (\equiv_{\mathrm{N}}\text{-refl}) \qquad \frac{\nu_1 \equiv_{\mathrm{N}} \nu_2 \qquad \nu_2 \equiv_{\mathrm{N}} \nu_3}{\nu_1 \equiv_{\mathrm{N}} \nu_3}\ (\equiv_{\mathrm{N}}\text{-trans}) \qquad \frac{\nu_1 \equiv_{\mathrm{N}} \nu_2}{\nu_2 \equiv_{\mathrm{N}} \nu_1}\ (\equiv_{\mathrm{N}}\text{-sym})$$

$$\frac{}{\{\![]\!\} \equiv_{\mathrm{N}} \{\![]\!\}}\ (\equiv_{\mathrm{N}}\text{-}\{\![]\!\}) \qquad \frac{\nu_1 \equiv_{\mathrm{N}} \nu_2 \qquad \nu_1^{\#} \equiv_{\mathrm{N}} \nu_2^{\#}}{\{\![\nu_1]\!\} +_N^{\#} \nu_1^{\#} \equiv_{\mathrm{N}} \{\![\nu_2]\!\} +_N^{\#} \nu_2^{\#}}\ (\equiv_{\mathrm{N}}\text{-}+_N^{\#})$$

$$\frac{\nu_1^{\#} \equiv_{\mathrm{N}} \nu_2^{\#}}{\nu_1^{\#} \rightarrow \nu \equiv_{\mathrm{N}} \nu_2^{\#} \rightarrow \nu}\ (\equiv_{\mathrm{N}}\text{-}\rightarrow_1) \qquad \frac{\nu_1 \equiv_{\mathrm{N}} \nu_2}{\nu^{\#} \rightarrow \nu_1 \equiv_{\mathrm{N}} \nu^{\#} \rightarrow \nu_2}\ (\equiv_{\mathrm{N}}\text{-}\rightarrow_2)$$

$$\frac{f \in \mathscr{F} \qquad i \in [\![1\,;|f|_{\mathscr{F}}]\!] \qquad \nu_i \equiv_{\mathrm{N}} \nu_i'}{f(\nu_1, \ldots, \nu_i, \ldots, \nu_{|f|_{\mathscr{F}}}) \equiv_{\mathrm{N}} f(\nu_1, \ldots, \nu_i', \ldots, \nu_{|f|_{\mathscr{F}}})}\ (\equiv_{\mathrm{N}}\text{-}\mathscr{F})$$

Si $\nu_1 \equiv_{\mathrm{N}} \nu_2$, on dit que ν_1 et ν_2 sont équivalents.

Néanmoins, on se rend compte que cette équivalence n'est rien d'autre que l'égalité sur N :

Lemme 2.3.3.17
Deux types normalisés sont équivalents *ssi* ils sont égaux.

Démonstration.
Le sens indirect est trivial.
Le sens direct se montre sans encombre par induction structurelle. □

En fait, une partie du travail est déléguée aux substitutions et l'autre à l'égalité sur les multi-ensembles. En expliciter la représentation — en Coq par exemple — pourrait rendre la chose moins aisée. Le temps a manqué pour cela.

Quoi qu'il en soit, nous pouvons maintenant parler d'unification pour les types normalisés :

Définition 2.3.3.18 (unificateur de types normalisés)
Une substitution de types normalisés θ est un unificateur de deux types normalisés ν_1 et ν_2 *ssi* :

$$\hat{\theta}(\nu_1) = \hat{\theta}(\nu_2)$$

Définition 2.3.3.19 (types normalisés unifiables)
Deux types normalisés ν_1 et ν_2 sont unifiables, noté $\nu_1 \sim_{\mathrm{N}} \nu_2$, *ssi* ils admettent un unificateur.

Exemple 2.3.3.20
Les types normalisés suivants sont unifiables :

— $\{\![int, float]\!\} \rightarrow int \sim_{\mathrm{N}} \{\![float]\!\} \rightarrow \alpha$ avec $\{\alpha \mapsto \{\![int]\!\} \rightarrow int\}$;

— $\{\![int, \alpha]\!\} \rightarrow \{\![]\!\} \sim_{\mathrm{N}} \{\![int, int, int]\!\} \rightarrow \{\![]\!\}$ avec $\{\alpha \mapsto \{\![int, int]\!\}\}$;

— $\{\![\{\![\alpha, \beta]\!\} \rightarrow \alpha, \alpha, list(\beta)]\!\} \rightarrow \alpha \sim_{\mathrm{N}} \{\![list(\gamma), \delta, \{\![\gamma, \delta]\!\} \rightarrow \delta]\!\} \rightarrow \delta$
avec $\{\gamma \mapsto \beta, \delta \mapsto \alpha\}$.

Le théorème fondamental assure alors l'équivalence entre unifiabilité sur les types et unifiabilité sur les types normalisés.

Théorème 2.3.3.21
Deux types admettent un unificateur bien formé *ssi* leurs normalisées admettent un unificateur bien formé.

Démonstration.
Voir annexe. □

Chapitre 3

Indexation

Dans le chapitre précédent, nous avons présenté l'*unification modulo isomorphismes de types* et l'algorithme que nous utilisons. Un système de recherche de fonctions par types peut faire directement appel à cet algorithme : un type de l'environnement est accepté s'il est unifiable avec le type demandé, c'est-à-dire si les deux types admettent un unificateur.

Une question se pose alors : ce système passe-t-il à l'échelle ? Un écosystème OCaml, essentiellement les paquets OPAM, peut aisément contenir des centaines de milliers d'identificateurs de fonctions. Dans tout ce chapitre, nous considérons une session OPAM contenant principalement la bibliothèques `core`, représentant au total 1 638 855 identificateurs. Afin de se placer dans des conditions similaires, on pourra exécuter :

```
$ git clone https://github.com/Drup/dowsing dowsing
$ cd dowsing
$ opam switch create dowsing ocaml-base-compiler.4.12.0
$ opam install . --deps-only
$ opam install core
$ make
```

Le programme `dowsindex` est muni d'une commande `stats` qui, comme son nom l'indique, calcule des statistiques sur les types de l'environnement — pour plus de détails, voir le chapitre suivant. Invoquons-la ainsi :

```
$ ./dowsindex stats --no-index "int -> int -> int"
------------------------------------------------------------
measure        total time (ms)    avg. time (µs)    # unif.
------------------------------------------------------------
variable       4896.79            52.2994            93630
constructor    5150.43            3.805              1353597
tuple          854.756            5.16116            165613
other          119.108            4.57843            26015
------------------------------------------------------------
total time (s): 11.0211
total # unif.: 1638855
```

Le système affiche le nombre d'appels à l'unification et le temps d'unification total. Pour cette requête très simple, le programme passe typiquement plus d'une dizaine de secondes à unifier. Ce n'est pas satisfaisant.

On peut gagner deux ordres de grandeur en mémorisant les types avec lesquels on a déjà testé l'unification. Cela revient à appeler l'unification autant de fois qu'il y a de types uniques dans l'environnement. On peut le constater en exécutant les commandes :

```
$ ./dowsindex save
$ ./dowsindex stats "int -> int -> int"
------------------------------------------------------------
measure        total time (ms)    avg. time (µs)    # unif.
------------------------------------------------------------
variable       518.931            318.95             1627
constructor    63.6716            2.49009            25570
tuple          11.7421            2.85141            4118
other          0.940561           3.57628            263
------------------------------------------------------------
total time (s): 0.595286
total # unif.: 31578
```

Le nombre d'appels est réduit à 31 578, soit environ 0.6s d'unification. Voyons une requête plus difficile — mais toujours sans polymorphisme — :

```
$ ./dowsindex stats "int -> int -> int -> int -> int"
------------------------------------------------------------
measure        total time (ms)    avg. time (µs)    # unif.
------------------------------------------------------------
variable       58149.9            35740.6            1627
constructor    71.579             2.79933            25570
tuple          12.8505            3.12057            4118
other          0.865698           3.29163            263
------------------------------------------------------------
total time (s): 58.2352
total # unif.: 31578
```

Cette fois, on a toujours 31 578 appels mais sur près d'une minute. Ce n'est pas satisfaisant. Les techniques développées dans ce chapitre permettent de réduire le nombre d'appels à 107, pour un temps d'unification de l'ordre de la milliseconde.

L'unification est très coûteuse. Il y a deux moyens d'y remédier. Le premier consiste à améliorer l'algorithme d'unification. Néanmoins, celui que nous utilisons est déjà optimisé et assez complexe. Le second consiste à réduire le nombre d'appels à l'unification. C'est la voie que avons explorée durant ce stage.

Plus précisément, il s'agit de solliciter le moins possible l'algorithme d'unification sans compromettre la qualité des résultats. L'idée est de concevoir un test moins coûteux qui, en cas d'indécision, s'en remettra à l'unification. Ce test doit être *suffisamment discriminant* et *correct* vis-à-vis de l'unifiabilité : s'il est négatif, alors les types ne sont effectivement pas unifiables.

On peut raisonner par *condition nécessaire d'unifiabilité*, ou encore *critère d'unifiabilité*. Le premier objectif de ce stage était de trouver des critères d'unifiabilité satisfaisant les deux conditions. Pour cela, nous avons étudié la constitution de l'environnement à l'aide de *métriques* appropriées. Deux critères se sont imposés ; l'un porte sur la *tête* des types, l'autre sur la *queue*.

Ces critères reposent sur certains traits structurels des types qui peuvent être encodés et mémorisés dans un *index*. Cela induit une étape de pré-traitement des types de l'environnement — pour le moment assez coûteuse mais non optimisée. En pratique, cette étape en vaut largement la peine. L'index prend la forme d'un *trie* — ou *arbre préfixe* — adapté aux critères, dans la même veine que [21].

3.1 Métriques de types

Dans cette section, nous présentons trois *métriques* sur les types (normalisés). Il s'agit de mettre en évidence des traits communs potentiellement discriminants. Nous en avons expérimenté plusieurs en les intégrant à la commande `stats`.

L'unification permet d'instancier les types, c'est-à-dire de substituer une ou plusieurs variables. Il est donc logique d'observer la situation de ces dernières.

Dans quelles quantités les trouve-t-on ? Intuitivement, plus un type compte de variables, plus il est polymorphe et plus l'unification est difficile. A l'inverse, si un nombre important de types de l'environnement en comportent peu — ce que nous constaterons —, on peut espérer arriver à un *critère d'unification* fort.

Où les trouve-t-on ? Les deux critères que nous avons construits reposent sur la réponse à cette question. Les variables en *tête* — correspondant au type de retour — caractérisent les cas les plus difficiles : par l'isomorphisme (curry), on peut élever le nombre de paramètres en substituant par un type-flèche.

Insistons sur le pragmatisme de cette approche : ce qui importe, c'est qu'un critère donne de bons résultats sur l'environnement étudié, en présence de répartitions biaisées.

3.1.1 Nombre de variables uniques

La première métrique consiste à compter le *nombre de variables uniques* dans un type :

Définition 3.1.1.1 (variables d'un type normalisé)
L'ensemble des variables d'un type normalisé ν, noté $\mathrm{vars}(\nu)$, est défini inductivement par :

$$\begin{aligned}
\mathrm{vars}(\alpha) &= \{\alpha\} \\
\mathrm{vars}(\{\![\nu_1, \ldots, \nu_n]\!\}) &= \bigcup_{i=1}^{n} \mathrm{vars}(\nu_i) \\
\mathrm{vars}(\nu^{\#} \to \nu) &= \mathrm{vars}(\nu^{\#}) \cup \mathrm{vars}(\nu) \\
\mathrm{vars}(f(\nu_1, \ldots, \nu_{|f|_{\mathscr{F}}})) &= \bigcup_{i \in [\![1\,;|f|_{\mathscr{F}}]\!]} \mathrm{vars}(\nu_i)
\end{aligned}$$

Définition 3.1.1.2 (μ^1)
Le nombre de variables uniques d'un type normalisé ν, noté $\mu^1(\nu)$, est le cardinal de son ensemble de variables.

Exemple 3.1.1.3
$\mu^1(\{[\alpha, list(\beta), \{[]\} \to \beta]\} \to \alpha) = 2$

Voyons quelle est la constitution de l'environnement à l'aune de cette métrique. Invoquons donc `dowsindex` avec une requête volontairement simple :

```
$ ./dowsindex stats "int -> int -> int" --measure unique-vars
------------------------------------------------------------
measure    total time (ms)    avg. time (µs)    # unif.
------------------------------------------------------------
0          28.1541            1.48948           18902
1          108.248            15.2291           7108
2          373.408            108.015           3457
3          16.8808            11.8295           1427
4          7.12752            18.4174           387
5          4.09412            30.7829           133
6          0.967503           14.6591           66
7          0.668526           37.1403           18
8          0.645638           25.8255           25
9          0.179052           22.3815           8
10         0.176668           22.0835           8
11         0.143051           23.8419           6
12         1.55115            141.014           11
13         0.298738           42.6769           7
14         0.279188           39.884            7
16         0.0150204          15.0204           1
17         5.4822             2741.1            2
18         6.71101            1677.75           4
19         0.173092           173.092           1
------------------------------------------------------------
total time (s): 0.555204
total # unif.: 31578
```

Premier constat : près de 60% de la population ne possède pas de variable, plus de 20% n'en possède qu'une. Deuxième constat — auquel on s'attendait — : l'unification des types polymorphes est plus difficile, c'est-à-dire en moyenne beaucoup plus coûteuse.

Cette suprématie des types non polymorphes nous dit la chose suivante : un critère les discriminant pourrait réduire considérablement le nombre d'unifications.

3.1.2 Genre de la tête

La deuxième métrique regarde le type en *tête*. Nous le définissons ainsi :

Définition 3.1.2.1 (tête d'un type normalisé)

La tête d'un type normalisé ν, notée $\uparrow \nu$, est définie par :

$$\begin{aligned} \uparrow (\nu^{\#} \rightarrow \nu) &= \nu \\ \uparrow \nu &= \nu \end{aligned}$$

La bonne formation des types est ici cruciale : on n'a pas besoin de prendre récursivement la tête du membre droit d'une flèche puisque ce dernier n'est pas lui-même une flèche. Nous avons les garanties suivantes :

Lemme 3.1.2.2
La tête d'un type normalisé bien formé est bien formée et n'est pas une flèche.

Démonstration.
Trivial avec la définition de la tête. □

De façon symétrique, on peut définir la *queue* d'un type :

Définition 3.1.2.3 (queue d'un type normalisé)
La queue d'un type normalisé ν, notée $\downarrow \nu$, est définie par :

$$\begin{aligned} \downarrow (\nu^{\#} \rightarrow \nu) &= \nu^{\#} \\ \downarrow \nu &= \{[]\} \end{aligned}$$

Définition 3.1.2.4 (μ^2)
La métrique μ^2 est définie par :

$$\mu^2(\nu) = [\uparrow \nu]$$

Exemple 3.1.2.5
On a les mesures :
— $\mu^2(\{[]\} \rightarrow \alpha) = \mathbf{var}$;
— $\mu^2(\{[int, int]\} \rightarrow list(\alpha)) = \mathbf{cons}_{list}$.

A nouveau, invoquons `dowsindex` avec la commande suivante :

```
$ ./dowsindex stats "int -> int -> int" --measure head
-----------------------------------------------------------
measure        total time (ms)    avg. time (µs)    # unif.
-----------------------------------------------------------
variable       506.137            311.086            1627
constructor    62.705             2.45229            25570
tuple          11.2598            2.73429            4118
other          0.815153           3.09944            263
-----------------------------------------------------------
total time (s): 0.580917
total # unif.: 31578
```

La catégorie « autre » regroupe les types que le système ne sait pas encore traiter : objets, classes, variants polymorphes… Y apparaît l'écrasante majorité des individus présentant un constructeur ou un uplet en tête : 94% de la population. Alors que nous savions déjà que 40% est dotée de polymorphisme, nous apprenons ici que 5% seulement arbore une variable en tête. Par ailleurs, le temps moyen d'unification sur cette sous-population est cent fois plus important que sur les autres : la présence d'une variable en tête caractérise manifestement les cas pathologiques.

3.1.3 Nombre de variables à la base

Symétriquement, on peut observer le nombre de variables dans la queue. Néanmoins, le plus intéressant est le nombre de variables dans la *base*, qui tient compte de la tête et de la queue :

Définition 3.1.3.1 (base d'un type normalisé)
La base d'un type normalisé ν, notée $\updownarrow \nu$, est définie par :

$$\updownarrow \nu = \downarrow \nu +_{\mathrm{N}}^{\#} \{\![\uparrow \nu]\!\}$$

Définition 3.1.3.2 (μ^3)
La métrique μ^3 est définie par :

$$\begin{aligned}
\mu^3(\{\![\nu_1, \ldots, \nu_n]\!\} \to \nu) &= \sum_{i=1}^{n} \mu^{3'}(\nu_i) + \mu^{3'}(\nu) \\
\mu^3(\alpha) &= 1 \\
\mu^3(\nu) &= 0 \\
\mu^{3'}(\alpha) &= 1 \\
\mu^{3'}(\nu) &= 0
\end{aligned}$$

Exemple 3.1.3.3
On a les mesures :
— $\mu^3(\{\![\alpha, int]\!\}) = 0$;
— $\mu^3(\{\![\alpha, int, \beta]\!\} \to \alpha) = 3$;
— $\mu^3(\{\![\{\![\alpha]\!\} \to \alpha, \alpha]\!\} \to list(\alpha)) = 1$.

Cette troisième métrique vient naturellement après les deux précédentes. Nous savons que la majorité des types possèdent peu ou pas de variables et qu'il est rare de les trouver en tête. Où sont-elles alors ? Nous poursuivons la recherche en élargissant le champ de vision à la base.

```
$ ./dowsindex stats "int -> int -> int" --measure spine-vars
-------------------------------------------------------
measure    total time (ms)    avg. time (µs)    # unif.
-------------------------------------------------------
0          70.9665            2.4932            28464
1          95.8333            44.6567           2146
2          397.937            486.476           818
3          5.00131            53.7775           93
4          1.3001             43.3366           30
5          0.28801            32.0011           9
6          0.00905991         4.52995           2
7          0.0109673          5.48363           2
8          0.0100136          5.00679           2
9          0.0100136          5.00679           2
10         0.0150204          7.51019           2
11         0.0143051          7.15256           2
12         0.014782           7.39098           2
```

```
13         0.0169277           8.46386            2
14         0.0171661           8.58307            2
--------------------------------------------------------
total time (s): 0.571445
total # unif.: 31578
```

Là encore, les proportions sont franches. Pas moins de 90% de la population ne présente pas de variables à la base. Le cas à deux variables, très minoritaire, semble particulièrement pathologique.

3.2 Critères d'unifiabilité

Dans cette section, nous présentons et formalisons deux critères d'unifiabilité dont l'intérêt potentiel est fondé sur les observations réalisées à l'aide des trois métriques.

Commençons par définir proprement ce qu'est un *critère d'unification correct* dans des termes proches de l'implémentation.

Définition 3.2.1 (booléens)
L'ensemble des booléens, noté $\mathbb{B}$, est défini par :

$$\frac{}{\mathbf{V} \in \mathbb{B}} \qquad \qquad \frac{}{\mathbf{F} \in \mathbb{B}}$$

Définition 3.2.2 (critère d'unifiabilité)
Un critère d'unifiabilité sur un domaine $\mathscr{D}$ est la donnée d'une fonction d'encodage de N dans $\mathscr{D}$ et une fonction de test de compatibilité de $\mathscr{D} \times \mathscr{D}$ dans $\mathbb{B}$. On note $\mathscr{C}_{\mathscr{D}}$ l'ensemble des critères sur $\mathscr{D}$.

Définition 3.2.3 (critère d'unifiabilité correct)
Un critère d'unifiabilité $(\mathit{encode}, \mathit{compat})$ est correct *ssi* :

$$\forall \nu_1 \in \mathrm{N}^{\mathbf{bf}},\ \forall \nu_2 \in \mathrm{N}^{\mathbf{bf}},\ \mathit{compat}(\mathit{encode}(\nu_1), \mathit{encode}(\nu_2)) = \mathbf{F} \implies \nu_1 \not\sim_{\mathrm{N}} \nu_2$$

Exemple 3.2.4
Le critère d'unifiabilité trivial sur N est correct : $(\nu \mapsto \nu, (\nu_1, \nu_2) \mapsto \mathbf{V})$.

Comme nous le verrons dans la section suivante, la fonction d'encodage sert précisément à encoder les types pour les stocker dans l'index tandis que la fonction de compatibilité teste rapidement l'unifiabilité sur les codes lors de la recherche.

3.2.1 Premier critère

Présentation

Le premier critère consiste à lier les genres des têtes des types à unifier. En effet, l'application d'une substitution préserve le genre de la tête — si ce n'est dans le cas d'une variable, mais avons vu que cela ne concerne qu'une minorité. On obtient la condition nécessaire suivante :

Théorème 3.2.1.0.1
Si deux types normalisés bien formés sont unifiables et que leurs têtes ne sont pas des variables, elles ont le même genre.

Démonstration.
Voir annexe. □

Exemple 3.2.1.0.2
Par contraposée, les types suivants ne sont pas unifiables :

— $\{[]\} \to int$ et $\{[]\} \to float$;

— $\{[]\} \to \{[]\}$ et $\{[]\} \to int$.

Implémentation

Cette condition nécessaire étant établie, exprimons-la à l'aide de la notion de critère que nous avons définie, c'est-à-dire dans les termes de l'implémentation.

Définition 3.2.1.0.3 (crit^1)
On note crit^1 le critère $(\mathrm{encode}^1, \mathrm{compat}^1)$ sur $\mathscr{D}^1 = \mathscr{G}$, où encode^1 et compat^1 sont définies par :

$$\begin{aligned}
\mathrm{encode}^1(\nu) &= [\uparrow \nu] \\
\mathrm{compat}^1(\mathbf{var}, g_2) &= \mathbf{V} \\
\mathrm{compat}^1(g_1, \mathbf{var}) &= \mathbf{V} \\
\mathrm{compat}^1(\mathbf{uplet}, \mathbf{uplet}) &= \mathbf{V} \\
\mathrm{compat}^1(\mathbf{fleche}, \mathbf{fleche}) &= \mathbf{V} \\
\mathrm{compat}^1(\mathbf{cons}_f, \mathbf{cons}_f) &= \mathbf{V} \\
\mathrm{compat}^1(g_1, g_2) &= \mathbf{F}
\end{aligned}$$

De façon plus visuelle, la figure 3.1 expose la chose sous la forme d'un arbre de décision auquel on soumet une requête. Les types de l'environnement sont distribués dans les sous-arbres en décidant quelle branche prendre à chaque nœud selon leur code. Les nœuds foncés correspondent aux sous-arbres compatibles avec la requête.

Théorème 3.2.1.0.4
Le critère d'unifiabilité crit^1 est correct.

Démonstration.
Par le théorème 5.2.1.1. □

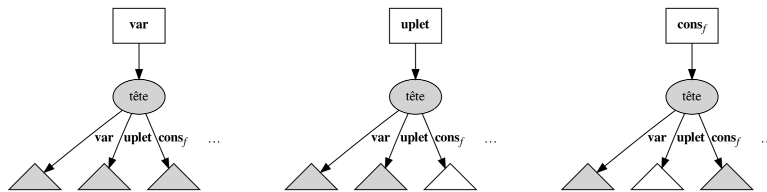

Figure 3.1

Apport

Voyons ce qu'apporte ce premier critère en pratique. Nous avons collecté dans la table 3.1 les résultats de plusieurs requêtes. Les deuxième et troisième colonnes donnent le temps total d'unification sans et avec le critère ; il en est de même pour les quatrième et cinquième avec le nombre d'unifications. On peut obtenir ces données à l'aide des commandes (il n'y a pas besoin de répéter la première) :

```
$ ./dowsindex save --index index.db --features head
$ ./dowsindex stats --index index.db --features head --with-features <type>
```

type	temps total (s)		nb. d'unifications	
	sans critère	avec critère	sans critère	avec critère
$int \rightarrow int \rightarrow int$	0.63	0.08 (12.1%)	31578	2714 (8.6%)
$int \rightarrow int \rightarrow int \rightarrow int$	1.08	0.88 (81.4%)	31578	2714 (8.6%)
$int \rightarrow (int \rightarrow int) \rightarrow list(\alpha)$	1.09	0.29 (26.4%)	31578	2945 (9.3%)
$int \rightarrow float \rightarrow bool \rightarrow \mathbf{unit}$	0.31	0.17 (53.6%)	31578	5745 (18.2%)
$\alpha \rightarrow int \rightarrow \mathbf{unit}$	0.83	0.30 (36.4%)	31578	5745 (18.2%)
$int \rightarrow int \rightarrow \alpha$	5.77	5.43 (94.1%)	31578	31578 (100%)

TABLE 3.1

Premier constat : le critère se comporte comme prévu. Deux types distincts mais dont les têtes ont le même genre unifient autant de fois et la requête présentant une variable en tête ne profite pas du critère.

Deuxième constat : pour la plupart des requêtes, ce premier critère assez simple diminue considérablement le nombre d'unifications — entre 81.8% et 91.4% sont éliminées. Tant que la requête n'a pas de variable en tête, on peut tirer pleinement parti du fait que 95% des types de l'environnement partagent cette caractéristique.

Le plus satisfaisant est de noter que le potentiel estimé à l'aide de la deuxième métrique se réalise par un critère intéressant. L'approche que nous avons adoptée en est corroborée.

3.2.2 Deuxième critère

Présentation

Le deuxième critère est plus sophistiqué que le premier. Observons plusieurs cas.

Plaçons-nous tout d'abord dans le cas où les deux types à unifier n'ont pas de variables à la base. Il vient aisément que, non seulement leurs têtes doivent avoir le même genre, mais on doit aussi trouver dans chaque queue les genres en quantités égales.

Plaçons-nous maintenant dans le cas où un seul des deux types possède cette caractéristique. Intuitivement, une variable de la base peut « absorber » des paramètres afin d'unifier — ce qui ne peut se produire dans un type en étant dépourvu. Ainsi, l'idée est que chaque genre non-variable représenté dans la queue d'un type quelconque doit nécessairement trouver son pareil au moins autant de fois dans la queue d'un type n'ayant pas cette capacité d'absorption.

Dans le dernier cas — le moins fréquent —, on ne peut rien affirmer.

Afin de formaliser la condition nécessaire résultante, on définit la notion de *multiplicité*, qui compte le nombre de types d'un genre donné dans la queue d'un type :

Définition 3.2.2.0.1 (multiplicité d'un genre dans un type normalisé)
La multiplicité d'un genre g dans un type normalisé ν, notée $\mu_g(\nu)$, est définie par :

$$\begin{aligned} \mu_g(\nu^\# \rightarrow \nu) &= \mu'_g(\nu^\#) && \\ \mu_g(\nu) &= 0 && \\ \mu'_g(\{\![\nu_1, \dots, \nu_n]\!\}) &= \sum_{i=1}^{n} \mu''_g(\nu_i) && \\ \mu''_g(\nu) &= 1 && \text{si } [\nu] = g \\ \mu''_g(\nu) &= 0 && \text{sinon} \end{aligned}$$

Introduisons également les *types normalisés simples*. Il s'agit des types ne possédant pas de variables à la base. Nous avons vu que neuf dixièmes des types de l'environnement sont simples. A cela s'ajoute le fait que le motif du matching est toujours simple.

Définition 3.2.2.0.2 (type normalisé simple)
Le prédicat sur les types normalisés « est simple », noté **simple**, est défini par :

$$\frac{[\nu] \notin \{\mathbf{var}, \mathbf{fleche}\}}{\nu \ \mathbf{simple}} \qquad \frac{\forall i \in [\![1\,;n]\!],\ [\nu_i] \neq \mathbf{var} \qquad [\nu] \neq \mathbf{var}}{\{\![\nu_1, \dots, \nu_n]\!\} \rightarrow \nu \ \mathbf{simple}}$$

Bien qu'intéressante dans le cadre des démonstrations, cette définition n'est pas adaptée à l'implémentation. Pour décider si un type est simple, on utilisera la métrique μ^3 :

Lemme 3.2.2.0.3
Un type normalisé est simple *ssi* sa mesure au sens de μ^3 est nulle.

Démonstration.
Les deux sens se démontrent sans difficulté. □

Ces définitions étant posées, nous sommes en mesure d'exprimer la seconde condition nécessaire :

Théorème 3.2.2.0.4
Si deux types normalisés bien formés ν_1 et ν_2 sont unifiables et ν_1 simple, la multiplicité de tout genre différent de **var** dans ν_1 est supérieure ou égale à celle dans ν_2.

Démonstration.
Voir annexe. □

Exemple 3.2.2.0.5
Par contraposée, les types suivants ne sont pas unifiables :

— $\{\![int, int]\!\} \rightarrow int$ et $\{\![int]\!\} \rightarrow float$;

— $\{\![int]\!\} \rightarrow int$ et $\{\![int, int, \alpha]\!\} \rightarrow \alpha$;

— $\{\![int, \{\![]\!\} \rightarrow \{\![]\!\}]\!\} \rightarrow \{\![]\!\}$ et $\{\![int, \{\![int]\!\} \rightarrow float, int]\!\} \rightarrow \alpha$.

Implémentation

Quant à l'implémentation, on peut d'abord exprimer le critère résultant pour chaque genre adéquat. De même que précédemment, on le représente visuellement par des arbres de décision en figure 3.2.

Définition 3.2.2.0.6 (crit_g^2)
Soit un genre de types normalisés g différent de **var** et **uplet**.
On note crit_g^2 le critère $(\mathrm{encode}_g^2, \mathrm{compat}_g^2)$ sur $\mathscr{D}_g^2 = \mathbb{B} \times \mathbb{N}$, où encode_g^2 et compat_g^2 sont définies par :

$$\begin{aligned}
\mathrm{encode}_g^2(\nu) &= (\text{vars-base}(\nu), \mu_g(\nu)) & \\
\text{vars-base}(\nu) &= \mathbf{F} & \text{si } \mu^3(\nu) = 0 \\
\text{vars-base}(\nu) &= \mathbf{V} & \text{sinon}
\end{aligned}$$

$$\begin{aligned}
\mathrm{compat}_g^2((\mathbf{V}, \mu_1), (\mathbf{V}, \mu_2)) &= \mathbf{V} & \\
\mathrm{compat}_g^2((\mathbf{V}, \mu_1), (\mathbf{F}, \mu_2)) &= \mathbf{V} & \text{si } \mu_1 \leqslant \mu_2 \\
\mathrm{compat}_g^2((\mathbf{V}, \mu_1), (\mathbf{F}, \mu_2)) &= \mathbf{F} & \text{sinon} \\
\mathrm{compat}_g^2((\mathbf{F}, \mu_1), (\mathbf{V}, \mu_2)) &= \mathbf{V} & \text{si } \mu_2 \leqslant \mu_1 \\
\mathrm{compat}_g^2((\mathbf{F}, \mu_1), (\mathbf{V}, \mu_2)) &= \mathbf{F} & \text{sinon}
\end{aligned}$$

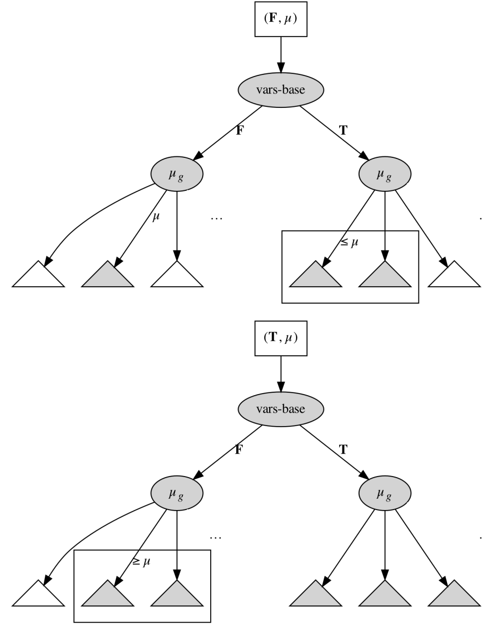

FIGURE 3.2

Théorème 3.2.2.0.7
Pour tout genre g différent de **var** et **uplet**, le critère d'unifiabilité crit_g^1 est correct.

Démonstration.
Par le théorème 5.2.2.1. □

Cependant, introduire autant de critères qu'il y a de constructeurs de types n'est pas une bonne idée. En fait, on peut « paralléliser » tous ces critères en travaillant sur des multi-ensembles de genres. Les inégalités ci-dessus laissent place à des inclusions entre multi-ensembles.

Définition 3.2.2.0.8 (crit^2)
On note crit^2 le critère (encode^2, compat^2) sur $\mathscr{D}^2 = \mathbb{B} \times \mathscr{G}^{\#}$, où encode^2 et compat^2 sont définies par :

$$\text{encode}^2(\nu) = \left(\text{vars-base}(\nu), \quad \biguplus^{\#}_{\mathscr{G}}{}_{g \notin \{\mathbf{var},\mathbf{uplet}\}} \; \biguplus^{\#}_{\mathscr{G}}{}_{[\![1\,;\mu_g(\nu)]\!]} \{\![g]\!\} \right)$$

$$\begin{aligned}
\text{compat}^2((\mathbf{V}, g_1^{\#}), (\mathbf{V}, g_2^{\#})) &= \mathbf{V} & & \\
\text{compat}^2((\mathbf{V}, g_1^{\#}), (\mathbf{F}, g_2^{\#})) &= \mathbf{V} & & \text{si } g_1^{\#} \subseteq^{\#}_{\mathscr{G}} g_2^{\#} \\
\text{compat}^2((\mathbf{V}, g_1^{\#}), (\mathbf{F}, g_2^{\#})) &= \mathbf{F} & & \text{sinon} \\
\text{compat}^2((\mathbf{F}, g_1^{\#}), (\mathbf{V}, g_2^{\#})) &= \mathbf{V} & & \text{si } g_2^{\#} \subseteq^{\#}_{\mathscr{G}} g_1^{\#} \\
\text{compat}^2((\mathbf{F}, g_1^{\#}), (\mathbf{V}, g_2^{\#})) &= \mathbf{F} & & \text{sinon}
\end{aligned}$$

Théorème 3.2.2.0.9
Le critère d'unifiabilité crit^2 est correct.

Démonstration.
On interprète l'inclusion sur les multi-ensembles en termes de multiplicités. Le résultat se déduit du théorème 3.2.2.0.7. □

Apport

Allions les deux critères. Les résultats sont présentés en table 3.2. Ils ont été obtenus grâce à la commande suivante :

```
$ ./dowsindex stats --with-features <type>
```

type	temps total (s)		nb. d'unifications	
	sans critères	avec critères	sans critères	avec critères
$int \to int \to int$	0.638	0.001 (0.23%)	31578	121 (0.38%)
$int \to int \to int \to int$	1.622	0.001 (0.08%)	31578	107 (0.34%)
$int \to (int \to int) \to list(\alpha)$	11.193	0.011 (0.93%)	31578	141 (0.45%)
$int \to float \to bool \to \mathbf{unit}$	0.636	0.004 (0.60%)	31578	126 (0.40%)
$\alpha \to int \to \mathbf{unit}$	1.180	0.225 (19.06%)	31578	2443 (7.74%)
$int \to int \to \alpha$	5.097	1.566 (30.71%)	31578	3677 (11.64%)

Table 3.2

Force est de constater que les critères fonctionnent très bien sur ces requêtes. Les plus simples sont traitées de façon instantanée. Il est satisfaisant de constater

que l'unification sur un environnement de dizaines de millers de types se réduit dans ces cas à quelques centaines seulement.

Au vu de ces faibles résidus, on peut raisonnablement affirmer avoir épuisé les cas simples. Un troisième critère devrait se confronter à des types plus retors.

3.3 Utilisation d'un trie

Nous avons à présent tous les éléments pour construire un *index*. On est naturellement confronté au choix de la structure de données à employer. L'idée de Gabriel Radanne, inspirée par [21], est d'adapter la structure de *trie* aux critères.

Les tries servent à stocker efficacement une table associative où les clefs sont des mots sur un alphabet donné. Ce n'est pas le cas ici. Nous travaillons plutôt sur la concaténation des codes des deux critères, dans laquelle est contenue toute l'information nécessaire aux tests de compatibilité. Ces derniers compliquent d'ailleurs la chose : plusieurs branches peuvent être compatibles avec la requête (voir les figures 3.1 et 3.2). En somme, il faut revoir la structure.

En première approximation, notre structure consiste en un arbre dont les nœuds internes sont étiquetés par des critères et les feuilles par des ensembles de types. On la définit ainsi :

Définition 3.3.1 (tries de types normalisés)
L'ensemble des tries de types normalisés, noté $\mathscr{T}$, est défini inductivement par :

$$\frac{N \subseteq \mathrm{N}}{\mathbf{feuille}(N) \in \mathscr{T}} \qquad\qquad \frac{crit \in \mathscr{C}_{\mathscr{D}} \qquad fils \in \mathscr{F}(\mathscr{D}, \mathscr{T})}{\mathbf{noeud}_{\mathscr{D}}(crit, fils) \in \mathscr{T}}$$

Exemple 3.3.2
Avec nos deux critères, le trie vide s'écrit :

$$\mathbf{noeud}_{\mathscr{D}^1}(\mathrm{crit}^1, g \mapsto \mathbf{noeud}_{\mathscr{D}^2}(\mathrm{crit}^2, (b, g^{\#}) \mapsto \mathbf{feuille}(\{\})))$$

On les manipule à l'aide de deux fonctions. La première ajoute un type à un trie par un appel récursif sur le sous-arbre associé au code du type ajouté.

Définition 3.3.3 (trie-ajoute)
La fonction trie-ajoute est définie récursivement par :

$$\begin{aligned}
\text{trie-ajoute}(\mathbf{feuille}(N), \nu) &= \mathbf{feuille}(N \cup \{\nu\}) \\
\text{trie-ajoute}(\mathbf{noeud}_{\mathscr{D}}(crit, fils), \nu) &= \mathbf{noeud}_{\mathscr{D}}(crit, d \mapsto \text{trie-ajoute}'_{\mathscr{D}}(crit, fils, \nu, d))
\end{aligned}$$

$$\begin{aligned}
\text{trie-ajoute}'_{\mathscr{D}}((encode, _), fils, \nu, d) &= \text{trie-ajoute}(fils(d), \nu) && \text{si } d = encode(\nu) \\
\text{trie-ajoute}'_{\mathscr{D}}((encode, _), fils, \nu, d) &= fils(d) && \text{sinon}
\end{aligned}$$

La seconde calcule l'ensemble des types d'un trie compatibles pour tous les critères avec une requête donnée. Elle consiste en un appel récursif sur les sous-arbres compatibles avec la requête.

Définition 3.3.4 (trie-cherche)
La fonction trie-cherche est définie récursivement par :

$$\text{trie-cherche}(\textbf{feuille}(N), \nu) = N$$

$$\text{trie-cherche}(\textbf{noeud}_{\mathscr{D}}((encode, compat), fils), \nu) = \bigcup_{d \in \mathscr{D} | compat(d, encode(\nu)) = \mathbf{T}} \text{trie-cherche}(fils(d), \nu)$$

L'interface de implémentation OCaml correspondante est présentée ci-dessous. On y retrouve les fonctions d'encodage et de test de compatibilité des critères — `compute` et `compatible` — la structure arborescente avec feuilles et nœuds — `Leaf` et `Node` — ainsi que les fonctions d'ajout et de recherche — `add` et `iter_with`. La fonction `compare` définit un ordre total sur les critères dont `Node` a besoin pour ses structures de données. `empty` est le trie vide.

Sa simplicité est particulièrement appréciable. Il est aisé d'ajouter et expérimenter de nouveaux critères.

```
module type FEATURE = sig
  type t
  val compute : Type.t -> t
  val compare : t -> t -> Int.t
  val compatible : query:t -> data:t -> Bool.t
end

module type NODE = sig
  type t
  val empty : t
  val add : Type.t -> t -> t
  val iter_with : Type.t -> t -> Type.t Iter.t
end

module Leaf : NODE
module Node (Feat : FEATURE) (Sub : NODE) : NODE
```

Chapitre 4

`dowsindex`

Dans ce dernier chapitre, nous visitons les fonctionnalités du programme `dowsindex`. On pourra en trouver le code source sur le dépôt *github* `Drup/dowsing`. La version actuelle est compatible avec les systèmes d'exploitation Linux et macOS.

`dowsindex` met en application les techniques présentées dans le chapitre précédent. Avant toute chose, il faut donc créer un *index* auquel `dowsindex` se référera au moment de la recherche. La commande suivante fait l'affaire :

```
$ dowsindex save
```

L'index est entreposé dans un répertoire standard destiné aux données d'applications. On peut aussi indiquer explicitement où le sauvegarder :

```
$ dowsindex save --index <fichier>
```

On peut également préciser quels paquets OPAM inclure dans l'index — notamment afin de réduire le temps de construction. La command suivante a cet effet, mais aussi — si l'index existe déjà — celui de charger ou recharger les paquets désignés :

```
$ dowsindex save [--index <fichier>] <paquet>...
```

Ceci étant fait — et une bonne fois pour toutes —, on peut maintenant utiliser `dowsindex` pour sa fin première : recherche une fonction par son type. Si l'index a été créé dans le répertoire standard, il suffit de demander :

```
$ dowsindex search <type>
```

Sinon, il faut préciser le chemin de l'index :

```
$ dowsindex search --index <fichier> <type>
```

Si tout se passe bien, le programme affiche la liste des fonctions retenues avec leurs types. Pour ne garder que les n premières réponses, on peut employer l'option `-n` :

```
$ dowsindex search [--index <fichiers>] -n <n> <type>
```

L'espace de recherche peut être limité à certains paquets — c'est d'ailleurs le scénario le plus courant — :

```
$ dowsindex search [--index <fichiers>] [-n <n>] <paquet>... <type>
```

On peut par exemple rechercher une fonction de la bibliothèque standard testant l'appartenance à une liste :

```
$ ./dowsindex search stdlib "'a list * 'a -> bool"
Parsing.is_current_lookahead : 'a -> bool
Obj.magic : 'a -> 'b
List.mem : 'a -> 'a list -> bool
List.memq : 'a -> 'a list -> bool
ListLabels.mem : 'a -> set:'a list -> bool
ListLabels.memq : 'a -> set:'a list -> bool
Compenv.print_version_string : unit -> 'a
Compenv.print_standard_library : unit -> 'a
```

Mentionnons une dernière fonctionnalité. Par défaut, `dowsindex` fait du *matching* (voir chapitre 2). Pour les requêtes non polymorphes, cela revient à unifier. Pour les requêtes polymorphes, cela signifie que les variables sont chacune remplacées avant unification par une constante fraîche — n'apparaissant pas dans l'environnement — qui leur est propre. Sous cette nouvelle apparence, elles ne sont unifiables qu'avec des variables ; autrement dit, cela force le polymorphisme. C'est généralement le comportement souhaité. Néanmoins, l'utilisateur peut formuler sa requête sous la forme plus générale d'un *schéma de types*. Les variables polymorphes non instanciables y sont spécifiées avant le type, les deux étant séparés d'un point ; les autres variables libres du type seront alors instanciables. En résumé, `dowsindex` accepte la syntaxe suivante :

$$\mathrm{T} \mid \mathscr{V} \ldots \mathscr{V} . \mathrm{T}$$

On peut par exemple demander :

```
$ dowsindex search "a b. ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a"
```

Les deux variables étant explicitement liées, cela revient à :

```
$ dowsindex search "('a -> 'b -> 'a) -> 'a -> 'b list -> 'a"
```

En revanche, les variables de cette requête sont instanciables car non liées :

```
$ dowsindex search ". ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a"
```

Chapitre 5

Conclusion

Nous arrivons au terme de ce rapport. Retraçons ses grandes lignes. A partir d'une implémentation de l'*unification modulo isomorphismes de types*, nous avons amené, défini et démontré deux *critères d'unifiabilité*. Ces critères sont utilisés par `dowsindex` dans la construction d'un *index*. Ce travail permet de réduire significativement le nombre d'appels à l'unification. Les requêtes les plus simples sont traitées de façon instantanée.

Il y a encore à faire : terminer les preuves, définir d'autres critères visant les cas plus difficiles, prendre en compte des types OCaml plus sophistiqués... Plusieurs aspects de l'indexation mériteraient qu'on s'y arrête : récupération des fonctions de l'environnement OPAM, sauvegarde et chargement de l'index (pour le moment réalisés avec le module `Marshall`). `dowsindex` souffre particulièrement du fait que la qualification des types par rapport au module englobant est mal assurée ; par exemple, la fonction `Format.pp_print_int` est connue sous le type `formatter -> int -> unit`, et non pas `Format.formatter -> int -> unit`.

Quoi qu'il en soit, nous espérons avoir posé de solides bases au système `dowsindex`. Je remercie Gabriel Radanne et Laure Gonnord pour leur encadrement tout au long du stage. Je ne remercie pas le SARS-CoV-2 pour les conditions particulières dans lesquelles il s'est déroulé.

Annexe

Cette annexe est dévolue aux démonstrations des théorèmes de normalisation et des deux conditions nécessaires d'unifiabilité.

5.1 Normalisation

5.1.1 Bonne formation d'une normalisée

Théorème 5.1.1.1
La normalisée de tout type est bien formée.

Démonstration.
Par induction structurelle et par cas sur le type.

Cas α : trivial.

Cas **unit** : trivial.

Cas $\tau_1 \times \tau_2$:
Par hypothèse d'induction, $\text{norm}(\tau_1)$ et $\text{norm}(\tau_2)$ sont bien formés.
On en déduit le résultat par les lemmes 5.1.1.2, 5.1.1.3 et 5.1.1.4.

Cas $\tau_1 \to \tau_2$:
Par hypothèse d'induction, $\text{norm}(\tau_1)$ et $\text{norm}(\tau_2)$ sont bien formés.
Par le lemme 5.1.1.2, $\text{norm}'(\text{norm}(\tau_1))$ est presque bien formé.

Si $\text{norm}(\tau_2) = \nu_2^{\#} \to \nu_2$:
$\text{norm}(\tau_2)$ étant bien formé, $\nu_2^{\#}$ est presque bien formé, ν_2 est bien formé et n'est pas une flèche.
Par le lemme 5.1.1.3, la somme de $\text{norm}'(\text{norm}(\tau_1))$ et $\nu_2^{\#}$ est presque bien formée.
On en déduit le résultat attendu.

Si non : trivial.

Cas $f(\overline{\tau})$: trivial avec l'hypothèse d'induction.

□

Lemme 5.1.1.2
Si ν est bien formé, $\text{norm}'(\nu)$ est presque bien formé.

Démonstration.
Trivial avec le lemme 2.3.3.6. □

Lemme 5.1.1.3
La somme de deux uplets presque bien formés est presque bien formée.

Démonstration.
Trivial avec la définition de **pbf**. □

Lemme 5.1.1.4
Si $\nu^{\#}$ dans $\mathrm{N}^{\#}$ est presque bien formé, $\mathrm{norm}''(\nu^{\#})$ est bien formé.

Démonstration.
Trivial avec la définition de norm''. □

5.1.2 Préservation de la bonne formation par application d'une substitution bien formée

Théorème 5.1.2.1
Pour tout ν dans $\mathrm{N}^{\mathbf{bf}}$ et θ dans $\Theta^{\mathbf{bf}}$, $\hat{\theta}(\nu)$ est bien formé.

Démonstration.
Par induction structurelle sur ν.

Cas α : $\hat{\theta}(\alpha)$ est bien formé car θ l'est.

Cas $\nu = \{\![\nu_1, \ldots, \nu_n]\!\}$:
Par hypothèse d'induction, les $\hat{\theta}(\nu_i)$ sont bien formés.
Par le lemme 5.1.1.2, les $\mathrm{norm}'(\hat{\theta}(\nu_i))$ sont presque bien formés.
Par le lemme 5.1.1.3, leur somme l'est aussi.
Par l'absurde, supposons que cette somme est de cardinal 1.
ν étant bien formé, n est différent de 1.
Il existe donc i dans $[\![1\,;n]\!]$ tel que, pour tout j différent de i, $\mathrm{norm}'(\hat{\theta}(\nu_j))$ est le multi-ensemble vide.
Par les lemmes 5.1.2.2 puis 5.1.2.3, ces derniers ν_j sont tous le multi-ensemble vide.
Absurde car ν est bien formé.
Le cardinal de la somme est donc différent de 1, ce qui permet de conclure.

Cas $\nu^{\#} \to \nu$ avec $\nu^{\#} = \{\![\nu_1, \ldots, \nu_n]\!\}$:
Par hypothèse, $\nu^{\#}$ est presque bien formé donc les ν_i sont bien formés.
Par hypothèse d'induction puis le lemme 5.1.1.2, les $\mathrm{norm}'(\hat{\theta}(\nu_i))$ sont bien formés.
Par le lemme 5.1.1.3, leur somme $\hat{\theta}(\nu^{\#})$ est presque formée.
Par hypothèse d'induction, $\hat{\theta}(\nu)$ est bien formée.

Si $\hat{\theta}(\nu) = \nu^{\#'} \to \nu'$:
$\hat{\theta}(\nu^{\#})$ étant bien formé, $\nu^{\#'}$ est presque bien formé, ν' est bien formé et n'est pas une flèche.
Par le lemme 5.1.1.3, la somme de $\hat{\theta}(\nu^{\#})$ et $\nu^{\#'}$ est presque bien formée.
On en déduit le résultat attendu.

Si non : trivial.

Cas $f(\nu_1, \ldots, \nu_{|f|_{\mathscr{F}}})$: trivial avec l'hypothèse d'induction.

□

Lemme 5.1.2.2
Soit ν dans N. Si $\mathrm{norm}'(\nu)$ est le multi-ensemble vide, ν aussi.

Démonstration.
Trivial avec la définition de norm'. □

Lemme 5.1.2.3
Soient ν dans $\mathrm{N}^{\mathbf{bf}}$ et θ dans $\Theta^{\mathbf{bf}}$. Si $\hat{\theta}(\nu)$ est le multi-ensemble vide, ν aussi.

Démonstration.
Par induction structurelle sur ν.

Cas α :
Par définition de $\hat{\theta}$: $\hat{\theta}(\alpha) = \theta(\alpha) = \{\!\![\,]\!\!\}$.
Absurde car θ est bien formée.

Cas $\nu = \{\!\![\nu_1, \ldots, \nu_n]\!\!\}$:
Par hypothèse, ν est bien formé donc les ν_i le sont aussi.
Leur somme étant vide, les $\mathrm{norm}'(\hat{\theta}(\nu_i))$ sont nécessairement tous vides.
Par le lemme 5.1.2.3, les $\hat{\theta}(\nu_i)$ sont tous le multi-ensemble vide.
Par hypothèse d'induction, les ν_i sont donc tous le multi-ensemble vide.
Absurde car ν est bien formé.

Cas $\nu^{\#} \to \nu$: absurde.

Cas $f(\overline{\nu})$: absurde.

□

5.1.3 Equivalence des unifiabilités

Définition 5.1.3.1 (normalisée d'une substitution de types)
La normalisée d'une substitution de types σ, notée $\mathrm{norm}(\sigma)$, est la substitution de types normalisés définie par :

$$\forall \alpha \in \mathscr{V},\ (\mathrm{norm}(\sigma))(\alpha) = \mathrm{norm}(\sigma(\alpha))$$

Définition 5.1.3.2 (normalisée inverse d'un type normalisé)
La normalisée inverse d'un type normalisé ν, notée $\mathrm{inorm}(\nu)$, est définie inductivement par :

$$\begin{aligned}
\mathrm{inorm}(\alpha) &= \alpha \\
\mathrm{inorm}(\{\!\![\,]\!\!\}) &= \mathbf{unit} \\
\mathrm{inorm}(\{\!\![\nu_1, \ldots, \nu_n]\!\!\}) &= \bigtimes_{i=1}^{n} \mathrm{inorm}(\nu_i) \\
\mathrm{inorm}(\nu^{\#} \to \nu) &= \mathrm{inorm}(\nu^{\#}) \to \mathrm{inorm}(\nu) \\
\mathrm{inorm}(f(\nu_1, \ldots, \nu_{|f|_{\mathscr{F}}})) &= f(\mathrm{inorm}(\nu_1), \ldots, \mathrm{inorm}(\nu_{|f|_{\mathscr{F}}}))
\end{aligned}$$

Définition 5.1.3.3 (normalisée inverse d'une substitution de types normalisés)
La normalisée inverse d'une substitution de types normalisés θ, notée $\mathrm{inorm}(\theta)$, est la substitution de types définie par :

$$\forall \alpha,\ (\mathrm{inorm}(\theta))(\alpha) = \mathrm{inorm}(\theta(\alpha))$$

Théorème 5.1.3.4
Deux types admettent un unificateur bien formé *ssi* leurs normalisées admettent un unificateur bien formé.

Démonstration.

(⇒) Soient deux types τ_1 et τ_2 admettant un unificateur bien formé σ :

$$\hat{\sigma}(\tau_1) \equiv_{\mathrm{T}} \hat{\sigma}(\tau_2)$$

Par le lemme 5.1.3.5, on peut passer aux normalisées :

$$\mathrm{norm}(\hat{\sigma}(\tau_1)) = \mathrm{norm}(\hat{\sigma}(\tau_2))$$

Par le lemme 5.1.3.6, il vient que norm(σ) est un unificateur des normalisées de τ_1 et τ_2 :

$$(\widehat{\mathrm{norm}}(\sigma))(\mathrm{norm}(\tau_1)) = (\widehat{\mathrm{norm}}(\sigma))(\mathrm{norm}(\tau_2))$$

Par bonne formation de σ et le lemme 5.1.3.11, cet unificateur est bien formé.

(⇐) Soient deux types τ_1 et τ_2 tels que leurs normalisées admettent un unificateur bien formé θ :

$$\hat{\theta}(\mathrm{norm}(\tau_1)) = \hat{\theta}(\mathrm{norm}(\tau_2))$$

Par bonne formation de θ et le lemme 5.1.3.9, il vient :

$$\mathrm{norm}((\widehat{\mathrm{inorm}}(\theta))(\tau_1)) = \mathrm{norm}((\widehat{\mathrm{inorm}}(\theta))(\tau_2))$$

Par le lemme 5.1.3.5, inorm(θ) est un unificateur de τ_1 et τ_2 :

$$(\widehat{\mathrm{inorm}}(\theta))(\tau_1) \equiv_{\mathrm{T}} (\widehat{\mathrm{inorm}}(\theta))(\tau_2)$$

Par bonne formation de θ et le lemme 5.1.3.12, cet unificateur est bien formé.

□

Lemme 5.1.3.5
Deux types sont équivalents *ssi* leur normalisées sont égales.

Démonstration.
Admis. □

Lemme 5.1.3.6
Pour tout σ dans $\Sigma^{\mathbf{bf}}$, on a :

$$\mathrm{norm} \circ \hat{\sigma} = \widehat{\mathrm{norm}}(\sigma) \circ \mathrm{norm}$$

Démonstration.
Admis. □

Lemme 5.1.3.7
La composée de inorm par norm est l'identité sur $\mathrm{N}^{\mathbf{bf}}$.

Démonstration.
Admis. □

Lemme 5.1.3.8
La composée de inorm par norm est l'identité sur $\Theta^{\mathbf{bf}}$.

Démonstration.
Soient θ dans $\Theta^{\mathbf{bf}}$ et α dans $\mathscr{V}$.
Par définition de norm puis inorm, il vient :

$$(\mathrm{norm}(\mathrm{inorm}(\theta)))(\alpha) = \mathrm{norm}(\mathrm{inorm}(\theta(\alpha)))$$

Par bonne formation de θ et le lemme 5.1.3.7, on obtient :

$$(\mathrm{norm}(\mathrm{inorm}(\theta)))(\alpha) = \theta(\alpha)$$

C'est bien le résultat attendu. □

Lemme 5.1.3.9
Pour tout θ dans $\Theta^{\mathbf{bf}}$, on a :

$$\hat{\theta} \circ \mathrm{norm} = \mathrm{norm} \circ \widehat{\mathrm{inorm}}(\theta)$$

Démonstration.
Par le lemme 5.1.3.12, inorm(θ) est bien formée car θ l'est.
Par le lemme 5.1.3.6, on a donc :

$$\mathrm{norm} \circ \widehat{\mathrm{inorm}}(\theta) = \widehat{\mathrm{norm}}(\mathrm{inorm}(\theta)) \circ \mathrm{norm}$$

Le lemme 5.1.3.8 nous donne le résultat attendu :

$$\mathrm{norm} \circ \widehat{\mathrm{inorm}}(\theta) = \hat{\theta} \circ \mathrm{norm}$$

□

Lemme 5.1.3.10
Une substitution de types σ est bien formée *ssi* :

$$\forall \alpha,\ \mathrm{norm}(\sigma(\alpha)) \neq \{[]\}$$

Démonstration.
Les deux sens se montrent aisément par l'absurde en invoquant le lemme 5.1.3.5. □

Lemme 5.1.3.11
La normalisée de toute substitution de types bien formée est bien formée.

Démonstration.
Soient σ dans $\Sigma^{\mathbf{bf}}$ et α dans $\mathscr{V}$.
Par définition de norm(σ), on a :

$$(\mathrm{norm}(\sigma))(\alpha) = \mathrm{norm}(\sigma(\alpha))$$

Par le lemme 5.1.1.1, cette dernière quantité est bien formée. Par bonne formation de σ et le lemme 5.1.3.10, elle est par ailleurs différente du multi-ensemble vide.
On en déduit la bonne formation de norm(σ). □

Lemme 5.1.3.12
La normalisée inverse de toute substitution de types normalisés bien formée est bien formée.

Démonstration.
Soient θ dans $\Theta^{\mathbf{bf}}$ et α dans $\mathscr{V}$.
Par bonne formation de θ et le lemme 5.1.3.8, on a :

$$\text{norm}((\text{inorm}(\theta))(\alpha)) = \text{norm}(\text{inorm}(\theta(\alpha))) = \theta(\alpha)$$

Par bonne formation de θ, cette dernière quantité est différente du multi-ensemble vide.
Le lemme 5.1.3.10 nous donne la bonne formation de inorm(θ). □

5.2 Conditions nécessaires d'unifiabilité

5.2.1 Première condition nécessaire

Théorème 5.2.1.1
Si deux types normalisés bien formés sont unifiables et que leurs têtes ne sont pas des variables, elles ont le même genre.

Démonstration.
Par hypothèse, il existe θ dans Θ telle que :

$$\hat{\theta}(\nu_1) = \hat{\theta}(\nu_2)$$

On passe à la tête dans cette égalité :

$$\uparrow \hat{\theta}(\nu_1) = \uparrow \hat{\theta}(\nu_2)$$

Le lemme 5.2.1.2 donne :

$$\uparrow \hat{\theta}(\uparrow \nu_1) = \uparrow \hat{\theta}(\uparrow \nu_2)$$

Soit i dans $\{1, 2\}$.
Par hypothèse, la tête de ν_i n'est pas une variable. Le lemme 5.2.1.3 nous assure donc que :

$$[\hat{\theta}(\uparrow \nu_i)] = [\uparrow \nu_1]$$

Par hypothèse, ν_i est bien formé. Par le lemme 3.1.2.2, sa tête n'est pas une flèche et donc, d'après la remarque précédente, $\hat{\theta}(\uparrow \nu_i)$ non plus. Par le lemme 5.2.1.4, on a ainsi :

$$[\uparrow \hat{\theta}(\uparrow \nu_i)] = [\hat{\theta}(\uparrow \nu_i)] = [\uparrow \nu_i]$$

On en déduit le résultat attendu. □

Lemme 5.2.1.2
Pour tout ν dans N et θ dans Θ, on a :

$$\uparrow \hat{\theta}(\nu) = \uparrow \hat{\theta}(\uparrow \nu)$$

Démonstration.
Par cas sur le type.
Seul le cas flèche est non trivial : $\nu^{\#} \to \nu$.
Par définition de $\uparrow \cdot$, on a d'abord :

$$\uparrow \hat{\theta}(\uparrow (\nu^{\#} \to \nu)) = \uparrow \hat{\theta}(\nu)$$

Si $\hat{\theta}(\nu) = \nu^{\#'} \to \nu'$:

Par définition de $\hat{\theta}$ et $\uparrow \cdot$, on a d'une part :

$$\uparrow \hat{\theta}(\nu^{\#} \to \nu) = \nu'$$

Par définition de $\uparrow \cdot$, on a d'autre part :

$$\uparrow \hat{\theta}(\nu) = \nu'$$

On en déduit le résultat.

Si non :

Par définition de $\hat{\theta}$ et $\uparrow \cdot$, on a :

$$\uparrow \hat{\theta}(\nu^{\#} \to \nu) = \hat{\theta}(\nu)$$

Le résultat se déduit du fait que $\hat{\theta}(\nu)$ n'est pas une flèche.

□

Lemme 5.2.1.3
L'application d'une substitution étendue à un type normalisé qui n'est pas une variable préserve son genre.

Démonstration.
Trivial avec la définition de l'extension d'une substitution. □

Lemme 5.2.1.4
La tête d'un type normalisé qui n'est pas une flèche préserve son genre.

Démonstration.
Trivial avec la définition de la tête. □

5.2.2 Deuxième condition nécessaire

Théorème 5.2.2.1
Si deux types normalisés bien formés ν_1 et ν_2 sont unifiables et ν_1 simple, la multiplicité de tout genre différent de **var** dans ν_1 est supérieure ou égale à celle dans ν_2.

Démonstration.
Soit g dans $\mathscr{G}$ différent de **var**.
Par hypothèse, il existe θ dans Θ telle que :

$$\hat{\theta}(\nu_1) = \hat{\theta}(\nu_2)$$

On passe aux multiplicités dans cette égalité :

$$\mu_g(\hat{\theta}(\nu_1)) = \mu_g(\hat{\theta}(\nu_2))$$

Par le lemme 5.2.2.3, on a d'une part :

$$\mu_g(\hat{\theta}(\nu_1)) = \mu_g(\nu_1)$$

Par le lemme 5.2.2.7, on a d'autre part :

$$\mu_g(\hat{\theta}(\nu_2)) \geqslant \mu_g(\nu_2)$$

On en déduit le résultat attendu. □

Lemme 5.2.2.2
Soient ν_1 et ν_2 dans N de même genre. Pour tout genre g, on a :

$$\mu''_g(\nu_1) = \mu''_g(\nu_2)$$

Démonstration.
Trivial avec la définition de μ''_g. □

Lemme 5.2.2.3
Soient ν dans $\mathrm{N}^{\mathbf{bf}}$ simple, g dans $\mathscr{G}$ différent de **var** et θ dans Θ. La multiplicité de g dans ν est la même que dans $\hat{\theta}(\nu)$.

Démonstration.
Par cas sur le type.
Seul le cas flèche est non trivial : $\nu^{\#} \to \nu$, avec $\nu^{\#} = \{\![\nu_1, \ldots, \nu_n]\!\}$.
Par hypothèse de simplicité et bonne formation, le genre de ν est différent de **var** et **fleche**.
Par le lemme 5.2.1.3, le genre de $\hat{\theta}(\nu)$ est donc différent de **fleche**.
Il vient alors par définition de $\hat{\theta}$:

$$\hat{\theta}(\nu^{\#} \to \nu) = \tilde{\theta}(\nu^{\#}) \to \hat{\theta}(\nu)$$

Par définition de μ_g et $\tilde{\theta}$, on a :

$$\mu_g(\hat{\theta}(\nu^{\#} \to \nu)) = \mu'_g\left(\mathop{\Large +}^{\#}_{\mathrm{N}}{}_{i \in [\![1\,;n]\!]} \mathrm{norm}'(\hat{\theta}(\nu_i))\right)$$

Par simplicité et bonne formation, tous les ν_i sont de genre différent de **var** et **uplet**.
Par le lemme 5.2.1.3, on a donc :

$$\forall i \in [\![1\,;n]\!],\ [\hat{\theta}(\nu_i)] = [\nu_i] \neq \mathbf{uplet}$$

Par définition de norm′, cela implique :

$$\forall i \in [\![1\,;n]\!],\ \mathrm{norm}'(\hat{\theta}(\nu_i)) = \{\![\hat{\theta}(\nu_i)]\!\}$$

On peut donc écrire :

$$\mu_g(\hat{\theta}(\nu^{\#} \to \nu)) = \mu'_g(\{\![\hat{\theta}(\nu_1), \ldots, \hat{\theta}(\nu_n)]\!\}) = \sum_{i=1}^{n} \mu''_g(\hat{\theta}(\nu_i))$$

Par le lemme 5.2.2.2, on a en outre :

$$\forall i \in [\![1\,;n]\!],\ \mu''_g(\hat{\theta}(\nu_i)) = \mu''_g(\nu_i)$$

Ce qui nous donne :

$$\mu_g(\hat{\theta}(\nu^{\#} \to \nu)) = \sum_{i=1}^{n} \mu''_g(\nu_i) = \mu'_g(\nu^{\#}) = \mu_g(\nu^{\#} \to \nu)$$

C'est bien le résultat attendu. □

Lemme 5.2.2.4
Pour tous $\nu_1^\#$ et $\nu_2^\#$ dans $\mathrm{N}^\#$ et g dans $\mathscr{G}$, on a :

$$\mu'_g(\nu_1^\# +_{\mathrm{N}}^\# \nu_2^\#) = \mu'_g(\nu_1^\#) +_{\mathrm{N}}^\# \mu'_g(\nu_2^\#)$$

Démonstration.
Trivial par définition de μ'_g. □

Lemme 5.2.2.5
Pour tout ν dans N de genre différent de **uplet** et g dans $\mathscr{G}$ différent de **var**, on a :

$$\mu'_g(\mathrm{norm}'(\hat{\theta}(\nu))) \geqslant \mu''_g(\nu)$$

Démonstration.
Par hypothèse, ν n'est pas un uplet. Reste trois cas.

Si ν est une variable : trivial.

Si non :
Par le lemme 5.2.1.3, $\hat{\theta}(\nu)$ est du même genre que ν, différent de **uplet** par hypothèse.
Par définition de norm′ et μ'_g, on a donc :

$$\mu'_g(\mathrm{norm}'(\hat{\theta}(\nu))) = \mu''_g(\hat{\theta}(\nu))$$

Le lemme 5.2.2.2 nous donne le résultat attendu :

$$\mu'_g(\mathrm{norm}'(\hat{\theta}(\nu))) = \mu''_g(\nu)$$

□

Lemme 5.2.2.6
Pour tout $\nu^\#$ dans $\mathrm{N}^\#$ presque bien formé et g dans $\mathscr{G}$ différent de **var**, on a :

$$\mu'_g(\tilde{\theta}(\nu^\#)) \geqslant \mu'_g(\nu^\#)$$

Démonstration.
Par définition de $\hat{\theta}$ et par le lemme 5.2.2.4, il vient :

$$\mu'_g(\tilde{\theta}(\nu^\#)) = \sum_{i=1}^{n} \mu'_g(\mathrm{norm}'(\hat{\theta}(\nu_i)))$$

$\nu^\#$ étant presque bien formé par hypothèse, tous les ν_i sont de genre différent de **uplet**.
Par le lemme 5.2.2.5, on a donc :

$$\forall i \in [\![1\,;n]\!],\ \mu'_g(\mathrm{norm}'(\hat{\theta}(\nu_i))) \geqslant \mu''_g(\nu_i)$$

D'où le résultat attendu :

$$\mu'_g(\tilde{\theta}(\nu^\#)) \geqslant \sum_{i=1}^{n} \mu''_g(\nu_i) = \mu'_g(\nu^\#)$$

□

Lemme 5.2.2.7
Pour tout ν dans $\mathrm{N}^{\mathbf{bf}}$ et g dans $\mathscr{G}$ différent de **var**, on a :

$$\mu_g(\hat{\theta}(\nu)) \geqslant \mu_g(\nu)$$

Démonstration.
Par cas sur le type.
Seul le cas flèche est non trivial : $\nu^{\#} \to \nu$ avec $\nu^{\#} = \{\!\![\nu_1, \ldots, \nu_n]\!\!\}$.

Si $\hat{\theta}(\nu) = \nu^{\#\prime} \to \nu'$:

Par définition de $\hat{\theta}$, μ_g et par le lemme 5.2.2.4, il vient :

$$\mu_g(\hat{\theta}(\nu^{\#} \to \nu)) = \mu'_g(\hat{\theta}(\nu^{\#})) +^{\#}_{\mathrm{N}} \mu'_g(\nu^{\#\prime}) \geqslant \mu'_g(\hat{\theta}(\nu^{\#}))$$

Si non :

Par définition de $\hat{\theta}$ et μ_g, il vient :

$$\mu_g(\hat{\theta}(\nu^{\#} \to \nu)) = \mu'_g(\hat{\theta}(\nu^{\#}))$$

Dans les deux cas, le lemme 5.2.2.6 nous donne le résultat attendu. □

Bibliographie

[1] Jacques HERBRAND. *Recherches sur la théorie de la démonstration.* fre. 1930. URL : http://eudml.org/doc/192791.

[2] M. GORDON et al. “A Metalanguage for Interactive Proof in LCF”. In : *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages.* POPL ’78. Tucson, Arizona : Association for Computing Machinery, 1978, p. 119-130. ISBN : 9781450373487. DOI : 10.1145/512760.512773. URL : https://doi.org/10.1145/512760.512773.

[3] Luis DAMAS et Robin MILNER. “Principal Type-Schemes for Functional Programs”. In : *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* POPL ’82. Albuquerque, New Mexico : Association for Computing Machinery, 1982, p. 207-212. ISBN : 0897910656. DOI : 10.1145/582153.582176. URL : https://doi.org/10.1145/582153.582176.

[4] Alberto MARTELLI et Ugo MONTANARI. “An Efficient Unification Algorithm”. In : *ACM Trans. Program. Lang. Syst.* 4.2 (avr. 1982), p. 258-282. ISSN : 0164-0925. DOI : 10.1145/357162.357169. URL : https://doi.org/10.1145/357162.357169.

[5] S. V. SOLOVIEV. “The category of finite sets and Cartesian closed categories”. In : *Journal of Soviet Mathematics* 22 (1983), p. 1387-1400.

[6] Andrew APPEL et David MACQUEEN. “Standard ML of New Jersey”. In : août 1991, p. 1-13. ISBN : 978-3-540-54444-9. DOI : 10.1007/3-540-54444-5_83.

[7] D. BERRY. “The Edinburgh SML Library”. In : 1991.

[8] Mikael RITTRI. “Using types as search keys in function libraries”. In : *Journal of Functional Programming* 1.1 (1991), p. 71-89. DOI : 10.1017/S095679680000006X.

[9] Kim B. BRUCE, Roberto DI COSMO et Giuseppe LONGO. “Provable isomorphisms of types”. In : *Mathematical Structures in Computer Science* 2.2 (1992), p. 231-247. DOI : 10.1017/S0960129500001444.

[10] Robero Di COSMO. “Type Isomorphisms in a Type-Assignment Framework”. In : *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* POPL ’92. Albuquerque, New Mexico, USA : Association for Computing Machinery, 1992, p. 200-210. ISBN : 0897914538. DOI : 10.1145/143165.143208. URL : https://doi.org/10.1145/143165.143208.

[11] Roberto Di Cosmo. "Deciding type isomorphisms in a type-assignment framework". In : *Journal of Functional Programming* 3.4 (1993), p. 485-525. DOI : 10.1017/S0956796800000861.

[12] Rittri, M. "Retrieving library functions by unifying types modulo linear isomorphism". In : *RAIRO-Theor. Inf. Appl.* 27.6 (1993), p. 523-540. DOI : 10.1051/ita/1993270605231. URL : https://doi.org/10.1051/ita/1993270605231.

[13] Sergei Soloviev. "A Complete Axiom System for Isomorphism of Types in Closed Categories." In : jan. 1993, p. 360-371.

[14] Roberto Di Cosmo. *Isomorphisms of Types : From Lambda-Calculus to Information Retrieval and Language Design.* CHE : Birkhauser Verlag, 1995. ISBN : 376433763X.

[15] Xavier Leroy. *The Caml Light system, documentation and user's guide Release 0.74.* Déc. 1997.

[16] Paliath Narendran, Frank Pfenning et Richard Statman. "On the unification problem for Cartesian closed categories". In : *Journal of Symbolic Logic* 62.2 (1997), p. 636-647. DOI : 10.2307/2275552.

[17] David Delahaye. "Information Retrieval in a Coq Proof Library Using Type Isomorphisms". In : *Types for Proofs and Programs.* Sous la dir. de Thierry Coquand et al. Berlin, Heidelberg : Springer Berlin Heidelberg, 2000, p. 131-147. ISBN : 978-3-540-44557-9.

[18] A. Boudet. "Competing for the AC-Unification Race". In : *Journal of Automated Reasoning* 11 (2004), p. 185-212.

[19] Xavier Leroy et al. *The Objective Caml system release 3.12 Documentation and user's manual.* Sept. 2010.

[20] Simon Marlow. "Haskell 2010 Language Report". In : (2010).

[21] Stephan Schulz. "Simple and Efficient Clause Subsumption with Feature Vector Indexing". In : *Automated Reasoning and Mathematics : Essays in Memory of William W. McCune.* Sous la dir. de Maria Paola Bonacina et Mark E. Stickel. Berlin, Heidelberg : Springer Berlin Heidelberg, 2013, p. 45-67. ISBN : 978-3-642-36675-8. DOI : 10.1007/978-3-642-36675-8_3. URL : https://doi.org/10.1007/978-3-642-36675-8_3.