Inférence de types et élaboration modulaire avec contraintes pour le langage ML étendu avec des abréviations de types

Carine Morel

▶ To cite this version:

Carine Morel. Inférence de types et élaboration modulaire avec contraintes pour le langage ML étendu avec des abréviations de types. Langage de programmation [cs.PL]. 2019. hal-02361707

HAL Id: hal-02361707
https://hal.inria.fr/hal-02361707

Submitted on 13 Nov 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Inférence de types et élaboration modulaire avec contraintes pour le langage ML étendu avec des abréviations de types

Carine Morel, sous la direction de Didier Rémy, INRIA Paris

28 août 2019

Introduction

Le contexte général

Les langages modernes polymorphes de haut niveau comme OCaml et Haskell ont des systèmes de type avancés qui permettent la détection d'expressions mal typées à la compilation. Pour conserver un typage au maximum implicite, ils s'appuient sur des algorithmes d'inférence de type, traditionnellement dérivés des algorithmes $\mathcal{J}$ et $\mathcal{W}$ de Milner ([7]) et basés sur l'unification de types et peuvent devenir complexes lorsque l'on étend le langage ML avec d'autres constructions (enregistrements, objets, sous-typage, GADT, polymorphisme de première classe etc..).

Inférer le type d'une expression se ramène à lui assigner une variable de type fraîche qui va être contrainte par la forme de l'expression mais aussi par son contexte. Par exemple, le type d'une abstraction est de la forme $\alpha \to \beta$ où α et β sont respectivement unifiées avec le type du paramètre de l'abstraction et celui de sa valeur de retour. C'est ensuite la résolution des contraintes ainsi générées puis la recontruction du terme élaboré qui résulte en une expression explicitement typée. L'inférence de types se décompose donc en trois étapes : génération de contraintes, résolution et élaboration.

Les algorithmes traditionnels mélangent génération et résolution. En plus de rendre les preuves plus compliquées, cela les rend moins modulaires : chaque extension du langage impose de refaire des parties de la résolution, alors qu'elle ne dépend pas du langage source. L'inférence de types par contrainte propose de résoudre ces problème en séparant les deux étapes : le programme est traduit en contraintes qui sont ensuite résolues. La résolution ne dépend alors que du langage de contraintes qui fait le lien entre les deux étapes. Cette méthode a été largement explorée dans l'article [9] pour le système de type de ML ou à la Hindley-Milner [3], en prouvant la décidabilité de l'inférence, la principauté et en montrant la robustesse de l'algorithme à l'ajout de diverses extensions du langage.

En pratique, cependant, l'inférence de types dans OCaml est toujours une version largement étendue de l'algorithme $\mathcal{W}$, i.e. qui mélangent les trois étapes. Dans le cadre d'une

refonte futur du typeur, de nombreux travaux, dont ce stage, s'attèlent à étudier la faisabilité de l'inférence par contrainte pour OCaml.

Le problème étudié

Mon travail de stage a consisté à étudier l'ajout d'une petite extension à l'inférence de types par contrainte de ML : les abréviations de type. Elles permettent l'introduction d'alias de types, potentiellement récursifs et sont utiles notamment dans les modules pour définir un type exigé par l'interface et pour les objets, où elles permettent d'avoir des types concis et lisibles. Elles modifient la résolution de l'égalité entre les types, puisqu'il peut être nécessaire d'expanser les abréviations pour les unifier mais aussi de mémoriser les expansions réalisées pour être efficace et permettre de mieux choisir les types affichés dans le terme élaboré. C'est une extension de langage courante et bien connue mais peu documentée.

Ce travail s'appuie sur le prototype Inferno ([8]) qui est une implémentation de l'inférence par contrainte pour le système de type à la Hindley-Milner. Il propose notamment une interface de solveur de contraintes qui permet à l'utilisateur de traiter en même temps la génération de contrainte et l'élaboration, minimisant ainsi le travail de maintenance en cas d'extensions du langage source.

L'objectif de ce stage est de proposer une formalisation de l'inférence de types par contrainte avec abréviations ainsi qu'une stratégie efficace de résolution des contraintes et une stratégie satisfaisante d'élaboration et d'implémenter le tout dans Inferno.

La contribution proposée

La difficulté de l'inférence avec les abréviations de type réside principalement dans trois points (1) la définition de nouveaux alias de type peut créer une multitude de cas de types pathologiques ; (2) le coût de l'expansion est potentiellement élévée ; (3) à l'élaboration, on dispose non plus d'un unique type solution mais d'un ensemble de types équivalents.

Pour résoudre ces problèmes, je me suis d'abord concentrée sur la définition de la bonne formation d'une abréviation, en particulier dans les cas de types récursifs. J'ai aussi traité le cas particulier des abréviations non productives, i.e. qui s'expansent complétement vers une variable de type et qui demandent un traitement particulier.

Pour résoudre les points (2) et (3), j'ai enrichi la structure des multi-équations des contraintes pour permettre la mémorisation des expansions réalisées et j'ai adapté les règles de génération et les règles de réécriture qui constituent l'algorithme d'unification et décrites dans [9] pour gérer les abréviations. La sémantique de la structure de multi-équations proposée est exprimée au travers d'invariants dont j'ai prouvé la préservation par les règles d'unification. Je décris aussi que les bonnes propriétés de cet algorithme (préservation des solutions et terminaison). Ensuite, je propose une stratégie d'application des règles d'unification pour essayer de minimiser les expansions réalisées. L'approche choisie permet à

l'utilisateur d'implémenter facilement sa propre stratégie d'élaboration des abréviations. Enfin, j'ai réalisé l'implémentation de l'algorithme d'unification [1] avec abréviations dans Inferno.

Les arguments en faveur de sa validité

L'algorithme d'unification proposée est intéressant car il reste proche de la version sans abréviation, utilisant ainsi les mêmes intuitions. Il permet de plus de laisser le choix de la stratégie d'élaboration à l'utilisateur. Sa validité est prouvé au travers de ses propriétés essentielles : chaque règle préserve les solutions des contraintes, l'algorithme termine et ces formes normales sont celles attendues. Enfin, le travail d'implémentation réalisé – même incomplet – montre que l'algorithme d'unification proposé est adaptable à Inferno.

Le bilan et les perspectives

Mon travail a été réalisé en partant de ML, c'est à dire d'un langage source minimal. Je suppose en effet que l'extension de l'inférence de type avec les abréviations est facilement composable avec d'autres. Il serait intéressant de vérifier cela formellement et en pratique. Les objets [10] ferait en particulier une bonne suite au travail présenté ici. En effet, les définitions de classe en OCaml génèrent automatiquement des abréviations pour représenter les types des objets de façon concise. Cette extension permettrait donc aussi de tester l'efficacité de l'algorithme d'unification avec abréviations.

Un autre travail intéressant à étudier dans la suite est le cas des conflits lors de l'inférence de types dont on ne parle pas du tout ici et en particulier la question de la génération de messages d'erreur. Elle a déjà été étudié dans le cas de l'inférence de type par contraintes [2] et il serait intéressant de considèrer quelle stratégie d'élaboration est alors pertinente et en quoi notre algorithme d'unification est compatible avec la génération de messages d'erreur.

1. Encore en cours de réalisation.

1 Langage source

Le langage source choisi est ML auquel on ajoute (1) des définitions d'abréviations de types, possiblement mutuellement récursives, et toutes placées au début du programme pour simplifier leur gestion (leur portée est tout le programme [2]) ; (2) des annotations de types (nécessaires pour utiliser les abréviations) ; (3) l'introduction explicite des nouvelles variables de type. Contrairement à ML, les variables de type ne sont pas liées implicitement : pour être utilisées dans une annotation, les variables de type doivent avoir été introduites auparavant.

1.1 Grammaire

$prog ::= \overline{typedef} \quad \overline{valdef}$	Programme
$typedef ::= \text{type } \overline{t\ (\overline{\alpha})\ = \tau}$	Définitions d'abréviations
$valdef ::= \text{let } x = expr$	Définition de valeur
$expr ::= c$	Littéral
$\mid x$	Variable
$\mid \text{fun var} \rightarrow expr$	Abstraction
$\mid expr\ expr$	Application
$\mid (expr, expr)$	Pair
$\mid \text{let } x = expr \text{ in } expr$	Let
$\mid (expr : \tau)$	Expression annotée
$\mid \text{new } \alpha \text{ in } expr$	Introduction d'une variable de type

où $\overline{t}$ signifie une séquence de t separés par des virgules lorsque t est un type, des "and" pour les définitions de type et par rien entre les différentes parties d'un programme.

On a choisi une syntaxe concrète minimale, proche de la syntaxe abstraite (AST) utilisée pour représenter ce langage. On introduit aussi du sucre syntaxique pour le placement des annotations :

$$
\begin{aligned}
\textbf{let } x : \tau_0 = e \left[\text{in } e'\right] &\equiv \textbf{let } x = (e : \tau_0) \left[\text{in } e'\right] \\
\textbf{let } (\textbf{new } \overline{\alpha})\ x = e &\equiv \textbf{let } x = \textbf{new } \overline{\alpha} \text{ in } e \\
\textbf{let } (\textbf{new } \overline{\alpha})\ x : \tau_0 = e &\equiv \textbf{let } x = \textbf{new } \overline{\alpha} \text{ in } (e : \tau_0)
\end{aligned}
$$

Comme les différentes syntaxes équivalentes ont la même représentation dans l'AST, il ne sera pas possible de reconstruire fidèlement (avec le sucre) à l'élaboration un programme. C'est cependant un problème orthogonal à celui traité ici et facile à résoudre en faisant des constructions primitives.

2. En Ocaml, la portée des abréviations est limitée mais on n'adresse pas ce problème ici.

Dans la suite, on prend les conventions de nommage suivantes : les variables de types sont nommées α ou β, x, y ou z désignent des variables de programme, F ou G des abréviations et τ un type. Enfin, dans le langage source, la convention de OCaml de préfixer les arguments est conservée bien que dans les notations formelles discutées par la suite, on préfère écrire $F\,\vec{\tau}$.

1.2 Syntaxe des types

On définit la syntaxe des types suivante : $\tau ::= \alpha \mid F\,\vec{\tau}$. Un type est soit une variable de type soit un type construit. Les constructeurs de type par défaut sont $\rightarrow$, $*$ et `int` et d'autres peuvent être ajoutés via les définitions d'abréviation. Ils définissent un environnement de type Γ. Les abréviations peuvent être récursives comme discuté en 2.5. De plus, on suppose fixé un ensemble dénombrable de variables de types, nommé $\mathcal{V}$.

1.3 Types clos

On définit les types clos comme l'ensemble des types sans variable de types. Un type clos est noté $\mathbf{t}$. Comme on a plusieurs constructeurs de types de base, dont `int` qui est d'arité 0, cet ensemble est non vide. On autorise en plus les types clos à être récursifs, et on utilise pour les représenter la syntaxe des types finis $\mu\alpha.\mathbf{t}$. Ce sont donc des arbres infinis réguliers. On suppose pour la suite que l'on sait vérifier une égalité entre de tels arbres.

2 Sur les abréviations

Les abréviations permettent d'introduire des alias de types. Peu de règles limitent leur définition en OCaml : abréviations non productives (par exemple, type $\alpha\,F = \alpha$, cf 2.2), mutuellement récusives (2.5), ou avec des paramètres fantômes (par exemple, type $(\alpha, \beta)\,F = \alpha \rightarrow \alpha$, cf 2.4) mais la bonne formation d'abréviations y est peu documentée. On donne ici une définition (incomplète) de la bonne formation des abréviations et de l'opération d'expansion (2.1) qui permet d'unifier des types avec les abréviations.

Par la suite, on écrit plus formellement une définition d'abréviation avec la syntaxe $F\,\vec{\alpha} \cong \tau$ qui décrit une abréviation F paramétrée par les variables de type $\vec{\alpha}$ et s'expansant vers τ. Les variables de types libres dans τ doivent être incluses dans $\vec{\alpha}$.

2.1 Expansion

Expanser une abréviation consiste à remplacer l'abréviation par son expansion, en substituant ses paramètres par les arguments de l'abréviation, c'est-à-dire, si $F\,\vec{\alpha} \cong \tau_0$, alors $F\,\vec{\tau}$ s'expanse vers $\tau_0\,[\vec{\alpha} \leftarrow \vec{\tau}]$. C'est une opération nécessaire lors de l'unification de deux types $F\,\vec{\tau} = G\,\vec{\tau}'$ dont les constructeurs de tête sont différents. Sans abréviation, il s'agit d'un conflit (unification impossible), mais avec des abréviations, il est nécessaire d'expanser

F et/ou G pour les unifier. Une première stratégie pourrait consister à expanser complétement les deux pour se ramener à une situation d'unification sans abréviation. Cependant, cette stratégie a deux défauts principaux : (1) elle peut être coûteuse et (2) elle limite l'élaboration. Expanser est une opération potentiellement coûteuse car le type expansé peut avoir une taille arbitrairement grande – pensons aux objets notamment. Elle peut aussi être inutile. Par exemple, avec les abréviations F et G définies telles que $F \cong \mathsf{int} * \mathsf{int} \to \mathsf{int}$ et $G \cong F$, l'expression `let y : G = x`, dans un contexte où x a type F, résulte en une égalité $F = G$. En expansant F puis G complétement, l'égalité devient $\mathsf{int} * \mathsf{int} \to \mathsf{int} = G$ puis $\mathsf{int} * \mathsf{int} \to \mathsf{int} = F$ et enfin $\mathsf{int} * \mathsf{int} \to \mathsf{int} = \mathsf{int} * \mathsf{int} \to \mathsf{int}$ alors qu'une unique expansion de G suffit à résoudre la contrainte. Si à la place de $\mathsf{int} * \mathsf{int} \to \mathsf{int}$, l'expansion de F est un terme de grande taille, cette méthode se montre inutilement très inefficace en temps et en espace.

La stratégie d'expansion agressive des abréviations limite aussi les stratégies d'élaboration possible. En effet, il est impossible de savoir quel est le type le moins expansé qui permet de réaliser une unification. L'exemple précédent illustre ce point : en expansant agressivement, le type retourné pour y est $\mathsf{int} * \mathsf{int} \to \mathsf{int}$, alors que le type F permet de signifier de façon plus concise l'égalité entre de les types de x et y, et pourrait donc être préférable. On va chercher par la suite à développer une stratégie de résolution des contraintes qui, contrairement à une stratégie d'expansion agressive, minimise les nombres d'expansions réalisées et mémorise les expansions réalisées pour permettre une meilleure élaboration .

On définit deux systèmes de réécriture liées à l'expansion. Le premier $\triangleright$ correspond à l'expansion en tête, utilisée dans la résolution de contraintes et la seconde $\rightsquigarrow$ représente l'expansion sous un contexte et est utile pour décrire l'égalité entre des types avec abréviations.

Expansion en tête On définit le système de réécriture $\triangleright$ correspondant à l'expansion *en tête* définie tel que $F\,\vec{\tau} \triangleright \tau_0\,[\vec{\alpha} \leftarrow \vec{\tau}]$ si et seulement si $F\,\vec{\alpha} \cong \tau_0$. Il est fortement normalisable sous les bonnes conditions sur les abréviations mutuellement récursives que (cf. 2.5). On définit aussi la clôture réflexive et transitive $\triangleright^*$ et l'expansion complète en tête $\triangleright^\infty$, telle que $\tau \triangleright^\infty \tau'$ si et seulement si $\tau \triangleright^* \tau' \not\triangleright$.

Les formes normales de $\triangleright$ sont uniques et sont soient des types dont le constructeur de tête n'est pas une abréviation, soit une variable de type. Ce dernier cas correspond aux abréviations non productives (voir 2.2).

Expansion sous contexte On note $\mathcal{E}$ un contexte de type, défini par la grammaire : $\mathcal{E} ::= [\,]\,|\,F(\tau_1, ..., \mathcal{E}, ...\tau_n)$.

Le système de réécriture $\rightsquigarrow$ est défini tel que $\tau_1 \rightsquigarrow \tau_2$ si et seulement s'il existe un contexte $\mathcal{E}$, $F\,\vec{\tau}$ et τ' tel que $\tau_1 = \mathcal{E}(F\,\vec{\tau})$ et $\tau_2 = \mathcal{E}(\tau')$ et $F\,\vec{\tau} \triangleright \tau'$. On explique dans 2.5 comment traiter les cas des abréviations récurisves pour que ce système soit rendu fortement normalisable, ce qui est une condition nécessaire pour que l'inférence termine.

2.2 Abréviations non productives

Les abréviations non productives sont un type d'abréviation qui nécessitent un traitement particulier. Une abréviation $F\,\vec{\alpha} \cong \tau$ est non productive si et seulement si $F\,\vec{\alpha} \triangleright^{\infty} \beta$. C'est le cas si $F\,\vec{\alpha} \cong \beta$ ou bien si $F\,\vec{\alpha} \cong G\,\vec{\tau}$ et G est non productive.

Pour la suite, on définit la fonction isprod qui s'applique à un constructeur de tête :

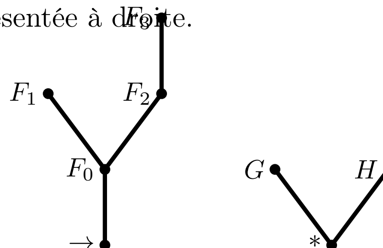
$$\mathsf{isprod}\,F = \begin{cases} \mathsf{Yes}\,G & \text{si } F\,\vec{\alpha} \triangleright^{\infty} G\,\vec{\tau} \\ \mathsf{No}\,i & \text{si pour tout } \alpha_1, ..., \alpha_n \text{ disjoints}, F\,(\alpha_1, ..., \alpha_n) \triangleright^{\infty} \alpha_i \end{cases}$$

Ces abréviations à première vue peu utiles peuvent cependant apparaître en OCaml au travers de modules ou de foncteurs. Comme elles s'expansent vers une variable de type, et non un type construit, elles nécessitent un traitement particulier dans nos algorithmes, comme nous l'expliquons dans la suite.

2.3 Représentation des abréviations

On peut voir un ensemble de définitions d'abréviations comme une forêt d'arbres dont les noeuds sont les constructeurs de types. Aux racines, on trouve des constructeurs de tête qui ne sont pas des abréviations. Les racines identiques sont partagées. Par exemple, les définitions d'abréviations de gauche donne la forêt représentée à droite.

$$\begin{array}{ll} F_0\,(\alpha, \beta) \cong \alpha \to \beta & F_1 \cong F_0\,(\mathsf{int}, \mathsf{int}) \\ F_2\,\alpha \cong F_0\,(\mathsf{int}, \alpha) & F_3\,\alpha \cong F_2\,\mathsf{bool} \\ G\,\alpha \cong \alpha * \alpha & H \cong \mathsf{int} * \mathsf{int} \end{array}$$



2.4 Décomposition et paramètres fantômes

La décomposition est une opération qui consiste à reporter une égalité sur deux types de la forme $F\,\vec{\tau}_1 = F\,\vec{\tau}_2$ ayant le même constructeur de tête F sur les arguments : $\vec{\tau}_1 = \vec{\tau}_2$.

Dans la définition d'une abréviation, il est possible d'utiliser des paramètres fantômes, c'est-à-dire des paramètres qui ne sont pas libres dans la définition de l'expansion. Par exemple, l'abréviation F définie comme par $F\ (\alpha, \beta) \cong \alpha * \alpha$ a une variable fantôme β. Dans le cas d'une décomposition sur F, il ne faut générer d'égalité sur son second paramètre.

On définit l'ensemble des paramètres fantômes d'une abréviation F et son complémentaire, l'ensemble des paramètres sur lesquels la décomposition doit être faite :

$$\begin{aligned} \mathsf{ph}\,(F) &= \{i \mid F\,(\alpha_1, ..., \alpha_n) \rightsquigarrow^{*} \tau \wedge \alpha_i \notin \mathsf{ftv}\,(\tau)\} \\ (\!|F|\!) &= \{1, ..., |F|\} \setminus \mathsf{ph}\,(F) \end{aligned}$$

Propriété 1. $i \in (\!|F|\!)$ *si et seulement si pour tout* $\vec{\alpha}$ *et pour tout* $\vec{\beta}$, $F\,\vec{\alpha} = F\,\vec{\beta} \Rightarrow \alpha_i = \beta_i$.

La décomposition donne alors $F\,\vec{\tau} = F\,\vec{\tau}\,'$ si et seulement si $\bigwedge_{i \in (\!| F |\!)} \tau_i = \tau_i'$.

2.5 Définitions valides et abréviations récursives

La définition d'abréviations récursives étant autorisée, on s'attèle ici à définir la bonne formation de telles abréviations, en s'assurant notamment de couvrir au moins les cas autorisés en OCaml avec l'option *-rectypes* (2.5.1). On remarque ensuite qu'elles peuvent faire boucler l'inférence de types et on propose une solution pour traiter ce problème (2.5.3).

2.5.1 Abréviations bien formées

Définition 1. *Un ensemble d'abréviations $(F_i\,\vec{\alpha}_i \cong \tau_i)^{i \in I}$ est* ***mutuellement récursif*** *si (et seulement si) pour tout i, j dans I, il existe un chemin d'expansions sous contexte et un contexte $\mathcal{E}_i^j$ tel que $F_i\,\vec{\alpha}_i \leadsto^* \mathcal{E}_i^j(F_j\,\vec{\tau})$ et $F_j\,\vec{\alpha}_j \leadsto^* \mathcal{E}_j^i(F_i\,\vec{\tau}\,')$.*

Définition 2. *Des définitions d'abréviations mutuellement récursives $(F_i\,\vec{\alpha}_i \cong \tau_i)^{i \in I}$ sont* ***régulières*** *si (et seulement si) toute occurence de F_j dans un des τ_i est de la forme $F_j\,\vec{\alpha}_j$.*

Un corolaire de cette définition est que les α_j sont tous identiques. Par exemple, les définitions mutuellement récursives suivantes sont régulières $F\,(\alpha, \beta) \cong G\,(\alpha, \beta) * \alpha$ et $G\,(\alpha, \beta) \cong F\,(\alpha, \beta)$.

Définition 3. *Des définitions d'abréviations mutuellement récursives $(F_i\,\vec{\alpha}_i \cong \tau_i)^{i \in I}$ sont dites* ***gardées*** *si (et seulement si) la relation $\{(i, j) \in I \times I \mid F_i\,\vec{\alpha}_i \rhd^* F_j\,\vec{\alpha}_j\}$ est acyclique.*

Cette dernière condition permet d'assurer la terminaison de $\rhd$.

Définition 4. *Des définitions d'abréviations mutuellement récursives sont bien formées si elles sont* ***gardées*** *et* ***régulières****.*

Voici un exemple d'abréviations mutuellement récursives bien formées :

```
type 'a tree = 'a * 'a forest and 'a forest = ('a tree) list
```

Des exemples de définitions d'abréviations qui ne passent pas nos restrictions :

```
type 'a foo = (('a*'a) foo) list (* Pas reguliere *)
type 'a foo = 'a bar  (* Pas gardee *)
and 'a bar  = 'a foo
```

L'exemple ci-dessous n'est pas non plus accepté avec nos définitions (il est considéré non régulier) alors que c'est une définition valide :

```
type 'a np = 'a
type 'a foo = ('a np) bar and 'a bar = ('a foo) -> ('a foo)
```

Pour l'accepter, il faudrait modifier la définition de la régularité pour prendre en compte les abréviations non productives.

2.5.2 Tri des abréviations

Pour pouvoir traiter correctement l'inférence de types avec les abréviations, et en assurer la terminaison, on réécrit les abréviations mutuellement récursives (2.5.3) en utilisant une version généralisée de la syntaxe finie des types récursifs $\mu\alpha.\ \tau$ grâce à des abréviations intermédiaires. Pour pouvoir faire ces réécritures efficacement, il faut dans un premier temps trier topologiquement les abréviations.

Les abréviations sont les triées en paquets d'abréviations mutuellement récursives et topologiquement, par ordre de dépendances, c'est-à-dire de sorte que si la définition d'une abréviation nécessite la définition d'une autre abréviation cette dernière est placée avant ou dans le même groupe (si elles sont mutuellement récursives). Par exemple,

```
type 'a l = int * 'a list
and 'a f = ('a t) l
and 'a e = string * 'a
and 'a t = 'a e * 'a f
```

est trié vers $\longrightarrow$

```
type 'a l = int * 'a list

type 'a e = string * 'a

type 'a f = ('a t) l
and  'a t = 'a e * 'a f
```

Une fois trié en paquet, on détermine l'ordre dans un paquet alphabétiquement (arbitrairement). On peut alors définir l'âge d'une abréviation F noté $\mathsf{age}(\mathrm{F})$ qui correspond au nombre d'abréviations définies après F. Ainsi, la première abréviation définie à pour âge le nombre totale de définitions d'abréviations et la dernière à un âge de 0. On a ainsi un ordre strict sur les abréviations : $F <_A G$ si et seulement si $\mathsf{age}(F) < \mathsf{age}(G)$.

2.5.3 Réécriture des abréviations mutuellement récursives

Le problème Prenons les 2 types mutuellement récursifs ainsi définis : $F\,\alpha \cong K\,(\alpha, G\,\alpha, F\,\alpha)$ et $G\ \alpha \cong K\,(\alpha, F\,\alpha, G\,\alpha)$ où K est un constructeur de type quelconque qui ne dépend ni de F ni de G. On essaie de résoudre la contrainte $F\,\alpha = G\,\alpha$.[3]

Définitions	$F\,\alpha \cong K\,(\alpha, G\,\alpha, F\,\alpha)$
	$G\,\alpha \cong K\,(\alpha, F\,\alpha, G\,\alpha)$
Contrainte initiale	$\beta_0 = F\,\alpha = G\,\alpha$
Expansion de F et G	$\beta_0 = F\,\alpha = K\,(\alpha, G\,\alpha, F\,\alpha) = G\,\alpha = K\,(\alpha, F\,\alpha, G\,\alpha)$
Décomposition de K	$\beta_0 = F\,\alpha = K\,(\alpha, \beta_1, \beta_2) = G\,\alpha\ \wedge\ \beta_1 = F\,\alpha = G\,\alpha\ \wedge\ \beta_2 = F\,\alpha = G\,\alpha$

Il n'y a aucun lien entre β_0, β_1 et β_2 et donc la résolution de la contrainte boucle.

3. Pour cet exemple et dans un soucis de simplicité, on utilise une stratégie d'expansion agressive mais pas à pas et avec mémorisation. Cette stratégie est ad-hoc : comme on conserve les termes expansés, on a toujours la possibilité d'expanser en boucle le même terme. On supposera donc que les termes expansés sont grisés et que seuls les termes non grisés peuvent encore être expansés.

Une première solution : le partage maximal Une première solution serait d'utiliser le partage maximal (hash consing) en mémorisant chaque association terme-variable. Dans l'exemple précédent, on aurait alors mémorisé $\beta = F\,\alpha$ et $\beta = G\,\alpha$ et l'expansion de $F\,\alpha$ et $G\,\alpha$ seraient $K\,(\beta, \alpha)$: pas de boucle donc. Cette solution créé des unifications qui ne sont pas liées à des contraintes du programme mais à l'égalité syntaxique entre deux termes. Dans notre cas, c'est gênant pour l'élaboration.

Deuxième solution : via une abréviation intermédiaire Une autre solution consiste à redéfinir les abréviations mutuellement récursives (qui sont détectées et groupées lors du tri) via des abréviations auxiliaires qui internalisent ce partage. Pour un ensemble de définitions mutuellement récursives et bien formées $(F_i\,\vec{\alpha} \cong \mathcal{E}_i(\vec{\alpha}, F_1\,\vec{\alpha}, ..., F_n\,\vec{\alpha}))^{i\in[1..n]}$, on définit les abréviations auxilaires $\left(F_i'(\vec{\alpha}, \vec{\beta}) \cong \mathcal{E}_i(\vec{\alpha}, \vec{\beta})\right)^{i\in[1..n]}$ où $\vec{\beta} = (\beta)^{i\in[1..n]}$ telles que abréviations récursives F_i se redéfinissent en $\left(F_i\,\vec{\alpha} \cong \text{ let rec}\,\vec{\beta} = \vec{F}'(\vec{\alpha}, \vec{\beta})\,\text{in}\,\mathcal{E}_i(\vec{\alpha}, \vec{\beta}\right)^{i\in I}$.

En termes de contrainte, une telle expansion se traduite par l'introduction de nouvelles variables de types β_i et par une conjonction d'équation de la forme $\beta_i = F_i'\,(\vec{\alpha}, \vec{\beta})$. La contrainte $\gamma_0 = F_i\,\vec{\alpha}$ après expansion de F_i devient donc $\exists\vec{\beta},\ \gamma_0 = F_i\,\vec{\alpha} = \mathcal{E}_i(\vec{\alpha}, \vec{\beta}) \bigwedge_{i\in[1..n]} \beta_i = F_i'\,(\vec{\alpha}, \vec{\beta})$. On n'a alors plus de récursion.

En reprenant l'exemple précédent, les nouvelles abréviations intermédiaires non récursives et les redéfinitions non récursives de F et G sont les suivantes :

$$F'\,(\alpha, \beta_F, \beta_G) = K\,(\alpha, \beta_G, \beta_F) \qquad G'\,(\alpha, \beta_F, \beta_G) = K\,(\alpha, \beta_F, \beta_G)$$

$$\begin{aligned} F\,\alpha &= \text{let rec}\,\beta_F = F'\,(\alpha, \beta_F, \beta_G)\,\text{in} \\ &\quad \text{let rec}\,\beta_G = G'\,(\alpha, \beta_F, \beta_G)\,\text{in}\,K\,(\alpha, \beta_G, \beta_F) \\ G\,\alpha &= \text{let rec}\,\beta_F = F'\,(\alpha, \beta_F, \beta_G)\,\text{in} \\ &\quad \text{let rec}\,\beta_G = G'\,(\alpha, \beta_F, \beta_G)\,\text{in}\,K\,(\alpha, \beta_F, \beta_G) \end{aligned}$$

Contrainte initiale	$\gamma_0 = F\,\alpha = G\,\alpha$
Expansions de F et G [4]	$\gamma_0 = F\,\alpha = K\,(\alpha, \beta_G, \beta_F) = G\,\alpha = K\,(\alpha, \beta'_F, \beta'_G) \wedge$ $\beta_F = F'\,(\alpha, \beta_F, \beta_G) \;\wedge\; \beta_G = G'\,(\alpha, \beta_F, \beta_G) \wedge$ $\beta'_F = F'\,(\alpha, \beta'_F, \beta'_G) \;\wedge\; \beta'_G = G'\,(\alpha, \beta'_F, \beta'_G) \wedge$
Décomposition sur K	$\gamma_0 = F\,\alpha = K\,(\alpha, \beta_G, \beta_F) = G\,\alpha \wedge$ $\beta_F = \beta'_G = F'\,(\alpha, \beta_F, \beta_G) = G'\,(\alpha, \beta'_F, \beta'_G) \wedge$ $\beta_G = \beta'_F = G'\,(\alpha, \beta_F, \beta_G) = F'\,(\alpha, \beta'_F, \beta'_G)$
Expansions de F'	$\gamma_0 = F\,\alpha = K\,(\alpha, \beta_G, \beta_F) = G\,\alpha \wedge$ $\beta_F = \beta'_G = F'\,(\alpha, \beta_F, \beta_G) = K\,(\alpha, \beta_G, \beta_F) = G'\,(\alpha, \beta'_F, \beta'_G) \wedge$ $\beta_G = \beta'_F = G'\,(\alpha, \beta_F, \beta_G) = F'\,(\alpha, \beta'_F, \beta'_G) = K\,(\alpha, \beta'_G, \beta'_F)$
Expansions de G'	$\gamma_0 = F\,\alpha = K\,(\alpha, \beta_G, \beta_F) = G\,\alpha \wedge$ $\beta_F = \beta'_G = F'\,(\alpha, \beta_F, \beta_G) = K\,(\alpha, \beta_G, \beta_F) = G'\,(\alpha, \beta'_F, \beta'_G) = K\,(\alpha, \beta'_F, \beta'_G) \wedge$ $\beta_G = \beta'_F = G'\,(\alpha, \beta_F, \beta_G) = K\,(\alpha, \beta_F, \beta_G) = F'\,(\alpha, \beta'_F, \beta'_G) = K\,(\alpha, \beta'_G, \beta'_F)$
Décompositions sur K	$\gamma_0 = F\,\alpha = K\,(\alpha, \beta_G, \beta_F) = G\,\alpha \wedge$ $\beta_F = \beta'_G = F'\,(\alpha, \beta_F, \beta_G) = K\,(\alpha, \beta_G, \beta_F) = G'\,(\alpha, \beta'_F, \beta'_G) \wedge$ $\beta_G = \beta'_F = G'\,(\alpha, \beta_F, \beta_G) = K\,(\alpha, \beta_F, \beta_G) = F'\,(\alpha, \beta'_F, \beta'_G)$

Il n'y a plus d'expansions possibles dans la contrainte finale : la résolution termine. En redéfinissant ainsi les abréviations mutuellement récursives, le système de réécriture $\triangleright$ termine. On suppose dans la suite que les abréviations récursives sont codés de cette façon : il n'y alors plus d'abréviations récursives.

Enfin, on remarquera qu'il n'est pas possible de faire apparaître les abréviations intermédaires dans une contrainte autrement que par une expansion (elles ne peuvent pas apparaître dans les annotations utilisateurs) et qu'il est donc possible de se ramener à l'élaboration aux définitions récursives initiales.

3 Génération de contraintes

La génération de contraintes consiste à traduire le programme en une contrainte sur les types. Pour cela, on définit un langage de contraintes (3.1) puis les règles permettant de traduire une contrainte de programme en une contrainte dans ce langage (3.2). Les explications sur la réduction du problème d'inférence de types à celui de résolution de contraintes, et notamment la question de la principauté des schémas de types peuvent être trouvés dans [9].

3.1 Syntaxe du langage des contraintes

Comme les définitions des abréviations sont placées en tête du programme, leur portée est globale et l'environnement de type Γ peut être construit indépendamment et en amont de la génération des contraintes. En conséquence, par la suite, on maintient Γ implicite pour ne pas surcharger l'écriture des contraintes. Notons aussi que les constructeurs de type stockés dans Γ contiennent toutes les informations utiles à la décomposition (notamment $(\!|F|\!)$, $\mathsf{isprod}\, F$ et le constructeur de type de l'expansion complète en tête pour calculer les incompatibilités entre abréviations) et à leur expansion, et sont précalculables avant la résolution des contraintes.

On exprime les contraintes dans le langage suivant [13] [8] :

$$\mathcal{C} ::= \mathsf{true} \mid \mathsf{false} \mid \epsilon \mid \mathcal{C} \wedge \mathcal{C} \mid \exists \alpha.\mathcal{C} \mid x\,\tau \mid (\lambda\alpha.\,\mathcal{C})\,\tau \mid \mathsf{let}\, x = \lambda\alpha.\,\mathcal{C}\,\mathsf{in}\,\mathcal{C}$$

En plus des contraintes habituelles (true, false, conjonction, existentielle sur les variables de types et multi-équation entre les types ϵ), le langage est enrichi pour pouvoir inférer le polymorphisme à la Hindley-Milner avec deux contraintes associées à (1) la généralisation (contrainte let) qui lie x à une contrainte abstraction [1] et (2) l'instanciation ($x\,\tau$ et $(\lambda\alpha.\,\mathcal{C})\,\tau$) de schémas de types qui est l'application d'une contrainte abstraction x ou $(\lambda\alpha.\,\mathcal{C})$ à un type τ.

3.2 Génération de contraintes

Le langage source nous permet de définir un programme comme une liste de définitions d'abréviations et une liste de définitions de valeurs. Les définitions d'abréviations sont utilisées pour déterminer l'environnement Γ. Les définitions de valeurs forment une contrainte de programme $[\![p]\!]$ qui est traduite vers le langage de contraintes avec les règles de réécriture données ci-dessous. Chaque définition de valeur est traduite individuellement et les contraintes générées sont liées par des contraintes letimbriquées comme suit :

$$[\![\emptyset]\!] \triangleq \mathsf{true} \qquad \text{[Programme vide]}$$

$$[\![(\mathsf{let}\, x = e) :: \mathrm{defs}]\!] \triangleq \mathsf{let}\, x = \lambda\alpha.\,[\![e : \alpha]\!]\,\mathsf{in}\,[\![\mathrm{defs}]\!] \qquad \text{[Programme non vide]}$$

C'est une définition inductive : chaque définition de valeur est traduite en une contrainte et passe sous l'abstraction à gauche d'une contrainte let. La contrainte droite est construite par récursion sur le reste du programme. S'il ne reste plus de définition, la contrainte droite est simplement true.

Exemple $\begin{array}{l}\mathsf{let}\, f = \lambda x.\,x \\ \mathsf{let}\, x = f\,1\end{array}$ $\xrightarrow{\text{génère la contrainte}}$ $\begin{array}{l}\mathsf{let}\, f = \lambda\alpha.\,[\![\lambda x.\,x : \alpha]\!]\,\mathsf{in} \\ \mathsf{let}\, x = \lambda\beta.\,[\![f\;\,1 : \beta]\!]\,\mathsf{in\,true}\end{array}$

3.2.1 Expressions sans annotation

Il n'y a pas de différences ici avec ce qui est proposé dans [8], excepté l'ajout des paires.

$$[\![x : \alpha]\!] \longrightarrow x\alpha \qquad \textbf{Var}$$
$$[\![(\lambda x.\, e) : \alpha]\!] \longrightarrow \exists\alpha_1\exists\alpha_2.\, (\mathsf{def}\, x = \alpha_1 \,\mathsf{in}\, [\![e : \alpha_2]\!] \wedge \alpha = \alpha_1 \to \alpha_2) \qquad \textbf{Abs}$$
$$[\![e_1\, e_2 : \alpha]\!] \longrightarrow \exists\beta_1\exists\beta_2.\, ([\![e_1 : \beta_2]\!] \wedge \beta_2 = \beta_1 \to \alpha \wedge [\![e_2 : \beta_1]\!]) \qquad \textbf{App}$$
$$[\![(e_1, e_2) : \alpha]\!] \longrightarrow \exists\alpha_1\exists\alpha_2.\, ([\![e_1 : \alpha_1]\!] \wedge [\![e_2 : \alpha_2]\!] \wedge \alpha = \alpha_1 * \alpha_2) \qquad \textbf{Pair}$$
$$[\![\mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 : \alpha]\!] \longrightarrow \mathsf{let}\, x = \lambda\beta.\, [\![e_1 : \beta]\!] \,\mathsf{in}\, [\![e_2 : \alpha]\!] \qquad \textbf{Let}$$

La contrainte def est le raccourci d'une contrainte let où l'abstraction décrit un type non généralisable : $\mathsf{def}\, x = \tau \,\mathsf{in}\, \mathcal{C} \triangleq \mathsf{let}\, x = \lambda\alpha.\, (\alpha = \tau) \,\mathsf{in}\, \mathcal{C}$.

3.2.2 Expressions avec annotations et introduction de variables de type

Pour pouvoir utiliser les abréviations, nous avons ajouté les annotations de types à notre langage source, ainsi que l'introduction de variables de types pour pouvoir maîtriser où une variable est liée. L'introduction de variables de type se traduit par une contrainte existentielle. Les annotations ajoutent une égalité entre deux types : le type annoté et la variable de type à inférer.

$$[\![(\mathbf{new}\ \alpha)\ e : \alpha]\!] \longrightarrow \exists\alpha.[\![e : \alpha]\!] \qquad \textbf{New}$$
$$[\![(e : \tau_0) : \alpha]\!] \longrightarrow [\![e : \alpha]\!] \wedge \alpha = \tau_0 \qquad \textbf{Annot}$$

Comme précedemment, les autres annotations sont du sucre syntaxique.

$$[\![(\lambda x : \tau_0.\, e) : \alpha]\!] \longrightarrow [\![(\mathsf{let}\, x = x : \tau_0 \,\mathsf{in}\, e) : \alpha]\!]$$
$$[\![\mathsf{let}\, x : \tau_0 = e_1 \,\mathsf{in}\, e_2 : \alpha]\!] \longrightarrow [\![\mathsf{let}\, x = e_1 : \tau_0 : \tau_0 \,\mathsf{in}\, e_2 : \alpha]\!]$$

4 Algorithme d'unification

Dans la partie précédente, nous avons montré comment le problème d'inférence de types est traduit en une contrainte. Dans [9], le solveur de contraintes proposé est découpé en deux parties : un algorithme d'unification, qui s'appuie sur l'égalité syntaxique pour résoudre les égalités entre les types, et un solveur pour gérer la généralisation et l'instanciation des variables de type. L'ajout d'abréviations change uniquement la question de l'unification : l'égalité syntaxique ne suffit pas pour décrire l'incompatiblité entre deux abréviations.

Dans la suite, on explique dans un premier temps la différence d'interprétation des contraintes avec et sans les abréviations (4.1) ; on décrit ensuite brièvement l'algorithme d'unification présenté dans [9] (4.2) puis les modifications effectuées pour unifier des types avec abréviations (4.3). Enfin, on donne les propriétés essentielles du système de réécriture utilisé pour décrire l'algorithme : terminaison, correction et formes normales (4.3.5).

L'unification se concentrant sur la résolution des équations entre types, le langage de contrainte utilisé dans la suite est un sous-ensemble de celui présenté en 3.1. On s'intéresse uniquement aux contraintes définies par la grammaire : $\mathcal{C} ::= \mathsf{true} \mid \mathsf{false} \mid \epsilon \mid \mathcal{C} \wedge \mathcal{C} \mid \exists \alpha.\mathcal{C}$.

4.1 Interprétation et satisfaisibilité des contraintes

On décrit une solution d'une contrainte $\mathcal{C}$ par une affection fermante ϕ qui envoie les variables de type $\mathcal{V}$ sur des types clos. Par homomorphisme, ϕ envoie aussi les types à des types clos. Une contrainte $\mathcal{C}$ est *satisfaisable* si elle a une solution ϕ, et on écrit $\phi \vdash \mathcal{C}$. Pour déterminer si une solution satisfait une contrainte, on définit l'interprétation d'une contrainte par le jugement inductif suivant :

$$\frac{}{\phi \vdash \mathsf{true}} \qquad \frac{\phi \vdash \mathcal{C}_1 \qquad \phi \vdash \mathcal{C}_2}{\phi \vdash \mathcal{C}_1 \wedge \mathcal{C}_2} \qquad \frac{\exists \mathbf{t}, \forall \tau \in \epsilon,\ \phi\tau \doteq \mathbf{t}}{\phi \vdash \epsilon} \qquad \frac{\phi\,[\alpha \leftarrow \mathbf{t}] \vdash \mathcal{C}_2}{\phi \vdash \exists \alpha.\ \mathcal{C}}$$

Sans abréviation, l'égalité $\doteq$ est simplement l'égalité syntaxique. Les abréviations introduisent de nouvelles équivalences dont il faut tenir compte pour résoudre une égalité entre deux types ; par exemple, pour les abréviations $F\,\alpha \cong \mathsf{bool} \to \alpha$ et $G\,\alpha \cong \alpha \to \mathsf{int}$, la contrainte $F\,\mathsf{int} = G\,\mathsf{bool}$ est toujours vraie alors que $F\,\mathsf{int}$ n'est pas syntaxiquement égal à $G\,\mathsf{bool}$. Ainsi, avec les abréviations, l'égalité $\doteq$ correspond à l'égalité modulo la relation d'équivalence induite par l'expansion d'abréviations. On la note alors $=_E$. Grâce aux propriétés des abréviations, il est possible de tester $\mathbf{t}_1 =_E \mathbf{t}_2$ en vérifiant que les types clos $\mathbf{t}_1$ et $\mathbf{t}_2$ s'expansent vers un type clos commun, i.e :

$$\mathbf{t}_1 =_E \mathbf{t}_2 \Leftrightarrow \exists \mathbf{t},\ \mathbf{t}_1 \rightsquigarrow^\infty \mathbf{t}\ \wedge\ \mathbf{t}_2 \rightsquigarrow^\infty \mathbf{t}$$

Définition 5. *Les contraintes $\mathcal{C}_1$ et $\mathcal{C}_2$ sont* ***équivalentes*** *si elles ont les mêmes solutions, i.e. ϕ satisfait $\mathcal{C}_1$ si et seulement si ϕ satisfait $\mathcal{C}_2$. On note cette relation $\mathcal{C}_1 \equiv \mathcal{C}_2$.*

On définit aussi $\mathcal{C}_1 \Rrightarrow \mathcal{C}_2$ si toute solution de $\mathcal{C}_1$ est solution de $\mathcal{C}_2$.

4.2 Algorithme d'unification sans abréviation

4.2.1 Lien entre contraintes et algorithme *Union find*

L'unification est le procédé de résolution d'équations entre termes, présenté initialement par Robinson [11]. Des algorithmes quasi linéaire en temps [6] [4] ont été proposés par la suite et s'appuient sur des structures de données qui résolvent le problème *union find* [12]. Ce sont des structures de données qui maintiennent des classes d'équivalences de variables et associent chaque classe à un descripteur.

Dans notre cas, sans abréviation, le descripteur vaut soit rien (la variable qui y est associée est généralisable) soit un type construit τ. Tant qu'elles ne sont pas résolues, les contraintes utilisées représentent un état intermédiaire de l'algorithme : par exemple, la contrainte $\alpha_1 = \alpha_2 = F\,\beta = G\,\gamma$ représente la classe d'équivalence contenant les variables

de type distinctes α_1 et α_2, associée au descripteur $F\,\beta$ et en attente de la mise à jour du descripteur en fonction du résultat de $F\,\beta = G\,\gamma$. La forme résolue d'une contrainte correspond à un état de l'algorithme *union find*.

Définition 6. *Une multi-équation $\alpha_1 = ... = \alpha_n = (F_i\,\vec{\tau}_i)^{i \in I}$ est en forme résolue si et seulement si les α_i sont distincts deux à deux et la taille de I est inférieure ou égale à 1, i.e. elle contient au plus un terme construit.*

Une contrainte est en forme résolue si et seulement si tous les existentiels sont en tête, chaque multi-équation est en forme résolue et chaque variable de type est membre d'au plus une multi-équation.

4.2.2 Règles d'unification

Comme dans [5], l'algorithme d'unification de [9] est présenté comme un système de réécriture. Les règles sont appliquées modulo α-conversion, modulo les permutations des membres d'une multi-équation, modulo commutativité et associativité des conjonctions et sous un contexte arbitraire.

$$(\exists \alpha. C_1) \wedge C_2 \xrightarrow{\alpha \notin \mathrm{ftv}(C_2)} \exists \alpha.(C_1 \wedge C_2) \qquad \textsc{ExAnd}$$

$$\alpha = \epsilon_1 \wedge \alpha = \epsilon_2 \longrightarrow \alpha = \epsilon_1 = \epsilon_2 \qquad \textsc{Merge}$$

$$\alpha = \alpha = \epsilon \longrightarrow \alpha = \epsilon \qquad \textsc{Stutter}$$

$$F\,\vec{\alpha} = F\,\vec{\beta} = \epsilon \longrightarrow \vec{\alpha} = \vec{\beta} \wedge F\,\vec{\alpha} = \epsilon \qquad \textsc{Decompose}$$

$$F\,(\vec{\tau}_1, \tau, \vec{\tau}_2) = \epsilon \xrightarrow{\tau \notin \mathcal{V} \,\wedge\, \alpha \notin \mathrm{ftv}(\vec{\tau}_1, \vec{\tau}_2, \epsilon)} \exists \alpha.\ F\,(\vec{\tau}_1, \alpha, \vec{\tau}_2) = \epsilon \ \wedge\ \alpha = \tau \qquad \textsc{Cut}$$

$$\alpha = F_0\,\vec{\tau}_0 = F_1\,\vec{\tau}_1 = \epsilon \xrightarrow{F_0 \neq F_1} \text{False} \qquad \textsc{Clash}$$

$$\tau \longrightarrow \mathsf{true} \qquad \textsc{Single}$$

$$\mathcal{C} \wedge \mathsf{true} \longrightarrow \mathcal{C} \qquad \textsc{True}$$

$$\mathcal{C} \xrightarrow{\text{Si le modèle est celui des termes et } \mathcal{C} \text{ est cyclique}} \mathsf{false} \qquad \textsc{Cycle}$$

$$C[\mathsf{false}] \xrightarrow{\text{Si}\, \mathcal{C} \neq []} \mathsf{false} \text{ où C est un contexte de contraintes.} \qquad \textsc{Fail}$$

où $\mathsf{ftv}\,(\mathcal{C})$ dénote les variables de types libres dans $\mathcal{C}$.

Une explication complète des règles peut être trouvée dans [9]. Appliquer complètement, ces règles emmènent les contraintes en forme résolue : la règle ExAnd sort les existentielles et permet d'obtenir une contrainte interne composée uniquement de conjonctions de multi-équations ; la règle Stutter assure que les variables soient toutes distinctes et la règle Merge que deux multi-équations ne partagent aucune variable ; enfin, les règles Decompose et Clash permettent d'obtenir un unique type construit par multi-équation. Sans la règle Cut, la règle Decompose s'écrit $F\,\vec{\tau}_0 = F\,\vec{\tau}_1 = \epsilon \longrightarrow \vec{\tau}_0 = \vec{\tau}_1 \wedge F\,\vec{\tau}_0 = \epsilon$ et alors $\vec{\tau}_0$ est copié. La règle Cut permet de restreindre l'utilisation de Decompose à des petits termes et de copier des variables de type plutôt que des types de profondeur arbitraire.

Ce système de réécriture des contraintes est fortement normalisant, préserve l'ensemble des solutions et ses formes normales sont soit false soit en forme résolue [9].

4.2.3 Quelques mots sur les contraintes let et instanciation

On a annoncé précédemment que l'ajout d'abréviations n'affectait que le problème d'unification et pas la phase suivante du solveur de contrainte qui gère la généralisation et l'instanciation. Pour ne pas rentrer dans les détails du solveur, on utilise ici la définition suivante de la contrainte let : $\mathsf{let}\, x = \lambda\alpha.\, C_1 \,\mathsf{in}\, C_2 \;\equiv\; \exists\alpha.\; C_1 \wedge C_2\,[x \leftarrow \lambda\alpha.\, C_1]$. La contrainte let assure donc que C_1 soit satisfaisable et décrit par quelle abstraction substituer x dans C_2. L'instanciation $(\lambda\alpha.\, \mathcal{C}_1)\,\tau$ se résout ensuite via une β-réduction.

La substitution copie la contrainte gauche $\mathcal{C}_1$ autant de fois qu'une contrainte d'instanciation de x (de la forme $x\,\tau$) apparaît dans $\mathcal{C}_2$. Pour limiter le coût en mémoire, mais aussi en temps en évitant de répéter plusieurs fois la résolution d'une même contrainte, il est important de faire l'unification sur $\mathcal{C}_1$ avant de faire les copies. En d'autres termes, la processus d'unification prend place dans la partie gauche d'un let et l'ajout d'abréviations n'y change rien.

4.3 Algorithme d'unification avec abréviations

Une première solution pour ajouter les abréviations à l'algorithme d'unification est de ne rien changer, et d'expanser les abréviations agressivement, soit à la génération de contraintes, soit à la place de la règle Clash : une multi-équation $F\,\vec{\alpha} = G\,\vec{\beta} = \epsilon$ où $F \neq G$ est si possible réécrite en remplaçant un des deux termes par son expansion ; s'il n'est pas possible de les expanser, la contrainte devient false (Clash). On peut aussi enrichir la structure de multi-équations pour mémoriser les expansions déjà réalisées afin de limiter le nombre d'expansions et d'utiliser les termes mémorisés à l'élaboration. Cependant, comme expliqué précedemment (voir 2.1), cette méthode réalise des expansions pas toujours nécessaires mais potentiellement coûteuses.

Nous proposons ici d'ajouter de la structure aux multi-équations pour mémoriser et retarder les expansions (4.3.1) et nous modifions les règles de réécriture en conséquence (4.3.3).

4.3.1 Structure des multi-équations

On veut pouvoir mémoriser les expansions réalisées et séparer abréviations productives et non productives pour permettre de les traiter différemment. Pour cela, on change le descripteur de la structure *union find* : au lieu d'être (1) rien ou (2) un type construit, le descripteur est composé de (1) un unique ensemble de termes non-productifs (potentiellement vide), ou de (2) deux ensembles : un productif qui mémorise les types productifs et un non productif pour les abréviations non productives.

On modifie suivant ce schéma les multi-équations en les partitionnant en :

— une unique vue non productive $[\delta]$ qui peut être vide

— un multi-ensemble de vues productives $(\nu_i)^{i \in I}$ non vides.

Les vues sont des multi-ensembles de termes construits. On note alors la multi-équation : $[\delta] = \langle \nu_0 \rangle = ... = \langle \nu_p \rangle = \epsilon$. Dans cette notation, ϵ contient au moins une variable, et peut contenir des termes qui ne sont pas dans des vues, que l'on nomme dans la suite *termes non traités* mais aussi des vues qui ne nous intéressent pas pour le point discuté. Les termes non traités nous permettent de conserver les règles de génération de contraintes précédentes (3.2). Ce sont de nouvelles règles d'unification (4.3.3) qui vont s'occuper de placer ces termes dans des vues et d'ajouter la structure susmentionnée dans la multi-équation.

Définition 7. *On définit le multi-ensemble des constructeurs de tête d'une vue ν, noté $\mathcal{HD}(v)$ tel que $\mathcal{HD}(F_1\,\vec{\tau}_1 = ... = F_n\,\vec{\tau}_n) = \{F_i \mid i \in [1..n]\}$.*

Définition 8. *Soit la multi-équation $\alpha_1 = .. = \alpha_n = [\delta] = (\langle \nu_i \rangle)^{i \in I} = (F_j\,\vec{\tau}_j)^{j \in J}$. Elle est en forme résolue si et seulement si (1) les α_i sont distincts deux à deux ; (2) J est vide, i.e. si elle n'a pas de termes non traités ; (3) la taille de I est inférieure ou égale à 1, i.e. elle a au plus une vue productive ; (4) la vue non productive δ et la vue productive ν sont telles que $\mathcal{HD}(\delta)$ et $\mathcal{HD}(\nu)$ sont des ensembles.*

4.3.2 Incompatibilités entre abréviations

Ci-dessus, la condition d'application de la règle Clash (voir 4.2) utilise la négation de l'égalité syntaxique. Cela ne fonctionne plus avec les abréviations. On définit en conséquence une relation d'incompatibilité entre deux constructeurs de tête F et G, noté $\bowtie_{\mathsf{max}}$.

Définition 9. $F \bowtie_{max} G \iff (\exists \vec{\alpha}, \vec{\beta}.\, F\,\vec{\alpha} = G\,\vec{\beta}) \equiv \mathit{false}$

Cette relation capture toutes les imcompatibilités entre abréviations : il n'est jamais nécessaire d'expanser un terme pour déclencher un conflits. On n'a cependant pas besoin d'une relation complète : on définit $\bowtie_{\mathsf{min}}$ la relation minimale autorisée, qui capture les conflits seulement sur les constructeurs de tête qui ne sont pas des abréviations.

Définition 10. $F \bowtie_{min} G \iff F \neq_S G$ *et F et G ne sont pas des abréviations.*

Cette relation correspond à une adaptation directe de celle utilisée dans la règle Clash de base. Une relation plus petite que celle-ci manquerait des cas de conflits. Dans la suite, on utilise la notation $\bowtie$ pour noter n'importe quelle relation qui satisfait $\bowtie_{\mathsf{min}} \subseteq \bowtie \subseteq \bowtie_{\mathsf{max}}$.

En termes d'implémentation, la relation $\bowtie_{\mathsf{max}}$ peut être coûteuse à calculer : il faut tester chaque paire d'abréviations. On peut cependant calculer en temps linéaire (sur le nombre de définitions d'abréviation) le résultat de la fonction isprod. On définit $\bowtie_{\mathsf{hd}}$ tel quel $F_0 \bowtie_{\mathsf{hd}} F_1 \iff \exists G_0, G_1,\, \mathsf{isprod}\,F_0 = \mathsf{Yes}\,G_0 \wedge \mathsf{isprod}\,F_1 = \mathsf{Yes}\,G_1 \wedge G_0 \neq G_1$. Pour cette relation, deux constructeurs de tête sont incompatibles s'ils s'expansent complétement en tête vers des constructeurs différents.

4.3.3 Règles d'unification

Les règles ExAnd, Stutter, Single, True, Cycle et Fail ne réécrivent pas les multi-équations et restent donc inchangées. Les autres sont remplacées par les suivantes :

$$\alpha = [\delta_1] = \epsilon_1 \;\wedge\; \alpha = [\delta_2] = \epsilon_2 \;\longrightarrow\; \alpha = [\delta_1 = \delta_2] = \epsilon_1 = \epsilon_2 \qquad \textbf{Merge}$$

$$F\,\vec{\tau} = \epsilon \xrightarrow{\mathsf{isprod}\,F\,=\,\mathsf{Yes}} \langle F\,\vec{\tau}\rangle = \epsilon \qquad \textbf{PDispatch}$$

$$[\delta] = F\,\vec{\alpha} = \epsilon \xrightarrow{\mathsf{isprod}\,F\,=\,\mathsf{No}\,i} [F\,\vec{\alpha} = \delta] = \alpha_i = \epsilon \qquad \textbf{NPDispatch}$$

$$F\,(\vec{\tau}_1, \tau, \vec{\tau}_2) = \epsilon \xrightarrow{\tau \notin \mathcal{V} \,\wedge\, \alpha \notin \mathrm{ftv}(\vec{\tau}_1, \vec{\tau}_2, \epsilon)} \exists\alpha.\, F\,(\vec{\tau}_1, \alpha, \vec{\tau}_2) = \epsilon \;\wedge\; \alpha = \tau \qquad \textbf{Cut}$$

$$\langle F\,(\vec{\tau}_1, \tau, \vec{\tau}_2) = \nu\rangle = \epsilon \xrightarrow{\tau \notin \mathcal{V}} \exists\alpha.\, \langle F\,(\vec{\tau}_1, \alpha, \vec{\tau}_2) = \nu\rangle = \epsilon \;\wedge\; \alpha = \tau \qquad \textbf{PCut}$$

$$\langle F\,\vec{\alpha} = \nu\rangle = \langle F\,\vec{\beta} = \nu'\rangle = \epsilon \;\longrightarrow\; \langle F\,\vec{\alpha} = F\,\vec{\beta} = \nu = \nu'\rangle = \epsilon \bigwedge_{i \in (\!|F|\!)} \alpha_i = \beta_i \qquad \textbf{Decompose}$$

$$\langle F\,\vec{\alpha} = F\,\vec{\beta} = \nu\rangle = \epsilon \;\longrightarrow\; \langle F\,\vec{\alpha} = \nu\rangle = \epsilon \qquad \textbf{PClean}$$

$$[F\,\vec{\alpha} = F\,\vec{\beta} = \nu] = \epsilon \;\longrightarrow\; [F\,\vec{\alpha} = \nu] = \epsilon \qquad \textbf{NPClean}$$

$$\langle F\,\vec{\tau} = \nu\rangle = \langle G\,\vec{\tau}' = \nu'\rangle = \epsilon \xrightarrow{F \bowtie G} \text{False} \qquad \textbf{Clash}$$

$$\langle F\,\vec{\alpha} = \nu\rangle = \langle \nu'\rangle = \epsilon \xrightarrow{F\,\vec{\beta} \cong G\,\vec{\tau}} \exists\vec{\beta}.\, \langle G\,\vec{\tau} = F\,\vec{\alpha} = \nu\rangle = \langle \nu'\rangle = \epsilon \bigwedge_{i \in (\!|F|\!)} \alpha_i = \beta_i \qquad \textbf{Expanse}$$

Comme la règle ExAnd est conservée, les règles considérées travaillent essentiellement sur des conjonctions de multi-équations. La règle Merge fusionne comme précédemment deux multi-équations partageant une variable. La multi-équation résultante a toujours une unique vue non productive, construite par union de celles des deux multi-équations fusionnées. Comme montré par la suite (voir 4.3.4), les termes de la vue non productive n'ont aucun rôle dans l'unification : ils ne servent que pour l'élaboration. On ne complique donc pas le système de réécriture avec une règle supplémentaire pour fusionner les vues non productives. Les règles PDispatch et NPDispatch permettent d'ajouter la nouvelle structure de vues dans une multi-équation. PDispatch place un terme productif dans une vue productive singleton. NPDispatch place un type non productif dans la vue non productive et le remplace dans la multi-équation par son expansion complète qui est une variable puisque le terme non productif est un petit terme. Pour ces abréviations, on emploie donc une stratégie d'expansion agressive qui est efficace (l'expansion d'une abréviation non productive réduit la taille du terme) ; cependant ce choix est au détriment de l'élaboration, puisqu'une partie de l'information – le chemin d'expansion – n'est pas calculé et mémorisé. La règle Cut (identique à celle du système initial) permet d'appliquer la règle NPDispatch en passant les termes non traités en petits termes. On ajoute aussi la règle PCut qui permet de passer les termes d'une vue productive en petits termes. Elle est indispensable pour la règle Expanse car l'expansion d'un petit terme productif n'est pas forcément un petit terme. Une règle NPCut est inutile puisque NPDispatch ne place que des petits termes dans la vue non productive.

Les comparaisons sur les termes non variable sont réalisées par les règles Decompose, Clash, PClean et NPClean. Decompose compare deux termes dans des vues productives différentes pour les fusionner. La règle fonctionne sur le même principe que précédemment : elle décompose deux termes ayant le même constructeur de tête et produit une conjonction d'équations sur leurs sous-termes décomposables. En plus, elle fusionne maintenant les deux vues contenant les termes décomposés. Contrairement à la règle de base, les deux termes décomposés sont conservés. En effet, il est possible que d'autres termes des deux vues fusionnées partagent un même constructeur de tête et il faut donc dans tous les cas une règle pour retirer les doublons. Il s'agit de la règle PClean. Elle compare les termes d'une même vue productive, alors que la règle NPClean travaille sur les vues non productive. Ces règles servent à être plus efficace en mémoire mais peuvent faire perdre de l'information pour l'élaboration. Par exemple, pour une abréviation F définie par $F(\alpha, \beta) \cong \alpha * \alpha$, la règle PClean retire un des deux termes dans la vue $\langle F(\mathsf{int}, \mathsf{int}) = F(\mathsf{int}, \mathsf{int} \to \mathsf{int})\rangle$, ce qui peut être restrictif à l'éboration. La règle Clash utilise désormais une relation d'incompatibilité $\bowtie$ (voir 4.3.2). En fonction de la relation utilisée, les conflits sont détectés plus ou moins tôt, i.e. après plus ou moins d'expansions. Enfin, la règle Expanse permet d'expanser en tête un terme d'une vue productive et ainsi de rendre de nouvelles décompositions possibles. Le terme expansé est grisé pour éviter de l'expanser de nouveau plus tard. Plus formellement, cela revient à considérer que la vue est découpée en deux parties : les termes qui ont déjà été expansés (grisés) et les autres. Toutes les règles précédentes s'appliquent aussi bien sur des termes grisés ou non. Seule la règle Expanse ne peut s'appliquer que sur un terme qui n'a pas été expansé, i.e. non grisé. Enfin, la règle ne s'applique que sur une multi-équation ayant au moins deux vues productives. Cette restriction permet d'assurer de ne pas continuer à expanser une vue alors que la multi-équation est en forme résolue.

4.3.4 Invariants

La structure des multi-équations décrites ci-dessus a un sens sémantique qu'on décrit ici au travers d'invariants que les règles de génération de contraintes et d'unification doivent préserver. Les preuves sont données dans l'appendix D. Notons que les invariants, bien que ne parlant que de multi-équations, sont valables sous un contexte de contraintes arbitraire.

Invariant 1. *La vue non productive est bien formée, i.e., tout type qu'elle contient est construit et son constructeur de tête est une abréviation non productive.*

Invariant 2. *Soit une multi-équation $[\delta] = \epsilon$, les éléments de la vue non productive ne contraignent pas les solutions de la multi-équation i.e. : $([\delta] = \epsilon) \equiv (\epsilon)$*

Ces invariants traduisent le fait que pour l'unification, toute l'information des termes d'une vue non productive est contenue dans la multi-équation, en dehors de la vue. En effet, la vue non productive ne contient que des termes qui ont été expansés complétement avant d'être placés dans la vue via la règle NPDispatch. La vue non productive est donc uniquement un espace de mémoire destiné à l'élaboration.

Invariant 3. *Les vues productives sont bien formées, i.e., tout type qu'elle contient est productif (i.e. il n'est ni non productif ni une variable de type) et les vues sont non vides.*

Invariant 4. *Soit une contrainte $\mathcal{C}$ et une de ses multi-équations $\langle\nu\rangle = \epsilon$. N'importe quel sous-ensemble non vide ν' de ν suffit à en définir les solutions dans $\mathcal{C}$, i.e. : $(\langle\nu'\rangle = \epsilon) \Rrightarrow (\langle\nu\rangle = \epsilon)$*

Une vue productive ne contient donc que des types équivalents.

Invariant 5. *Pour tous termes $F_i\vec{\tau}_i$ et $F_j\vec{\tau}_j$ dans une vue productive $\langle\nu\rangle$:*

a- soit il existe un chemin d'expansions en tête de F_i à F_j et toutes les abréviations intermédiaires expansées en tête sont dans $\mathcal{HD}(\nu)$

b- soit il existe un chemin d'expansions en tête de F_j à F_i et toutes les abréviations intermédiaires expansées en tête sont dans $\mathcal{HD}(\nu)$

c- soit il existe un constructeur de tête F_0 tel que F_i et F_j s'expansent en tête vers lui et toutes les abréviations intermédiaires sont dans $\mathcal{HD}(\nu)$.

On a décrit plus haut (cf 2.3) qu'un ensemble de définitions d'abréviations pouvait être représenté par une forêt d'arbres où les noeuds sont des abréviations et sont liés par la relation $\triangleright$. Dans cette représentation, l'invariant précédent stipule que les constructeurs de tête d'une vue forme un sous-arbre continu (sans trou) de cette forêt d'arbres, avec éventuellement plusieurs fois le même noeud. Par exemple, avec l'ensemble de définitions donné en 2.3, une vue respectant l'invariant pourrait être $[F_0\,\vec{\alpha}_0 = F_1\,\vec{\alpha}_1 = F_2\,\vec{\alpha}_2]$ alors que $[F_0\,\vec{\alpha}_0 = F_3\,\vec{\alpha}_3]$ ne le respecte pas. Informellement, l'invariant est préservé par PDISPATCH qui créé un arbre composé d'un unique noeud. PCLEAN retire un noeud doublon. EXPANSE ajoute un nouveau noeud qui est l'expansion en tête d'un noeud existant. Soit le nouveau noeud est un doublon, soit il est nouveau et l'arbre de la vue est prolongé vers le bas. Enfin DECOMPOSE prend deux sous-arbres chacun continu et qui partagent un même noeud et construit un nouvel arbre par union. Ce nouvel arbre est lui aussi continu puisque les deux arbres qui le forment partagent au moins un noeud.

4.3.5 Propriétés du système de réécriture des contraintes

Lemme 1. *Cet algorithme d'unification dispose de propriétés similaires à celui sans abréviation.*

(1) Les règles préservent les solutions : $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$ implique $\mathcal{C}_1 \equiv \mathcal{C}_2$.

(2) La réécriture termine.

(3) Une forme normale est soit false *soit de la forme $\exists\vec{\alpha}.\mathcal{C}$ où $\mathcal{C}$ est en forme résolue.*

5 Stratégie d'unification

Ci-dessus, on a défini les règles de réécriture pour simplifier les problèmes d'unification en présence d'abréviations. Nous définissons ici une stratégie pour répondre à notre objectif : limiter le nombre d'expansions pour être efficace et pour fournir une information plus pertinente pour l'élaboration.

On définit pour commencer les notions de rang d'une abréviation et de représentant d'une vue productive, deux concepts utiles pour notre stratégie, puis on introduit des règles composées qui combinent les règles élémentaires définies ci-dessus et les nouveaux invariants que l'utilisation des règles composées induit.

5.1 Rang et représentant

Définition 11. *Le rang d'une abréviation est le nombre maximal d'expansions possibles en tête.*

Par extension, le rang d'un type construit $F\,\vec{\tau}$ est le rang de son constructeur de tête. Le rang d'une abréviation nous permet notamment de déterminer quelle expansion réaliser en premier et de décrire des invariants de la vue productive par la suite. On définit aussi le représentant d'une vue comme son terme de plus petit rang. Son unicité constitue un des nouveaux invariants préservés par la stratégie d'unification décrites plus bas. On note le représentant $\mathcal{R}(\nu)$ et on définit le rang d'une vue productive comme étant le rang de son représentant.

5.2 La stratégie

Le système de réécriture défini par les règles d'unification (4.3.3) est non déterministe et certains chemins sont inefficaces. Par exemple, pour la multi-équation $\langle F\,\vec{\alpha} = F\,\vec{\beta}\rangle = \epsilon$, il est possible d'appliquer Expanse sur les deux termes de la vue productive puis de faire deux fois PClean , mais il est aussi possible d'appliquer d'abord PClean une fois et Expanse sur le terme restant. L'objectif de notre stratégie est de restreindre les chemins de réécriture pour limiter l'utilisation de la règle Expanse.

Pour amener les multi-équations dans une forme résolue (une vue productive au plus et pas de termes non traités), il faut fusionner autant que possible les multi-équations et leurs vues productives. La fusion de vues productives est réalisée par la décomposition et n'est possible que si les deux vues ont un constructeur de tête en commun. Comme on souhaite en limiter l'usage de la règle d'expansion, elle ne doit servir qu'à atteindre cette précondition (ou à déclencher un conflit). On ne veut donc jamais réaliser d'expansion quand une décomposition est possible puisque la décomposition est moins coûteuse et toujours nécessaire. Par exemple, il n'est pas nécessaire de faire d'expansion pour atteindre la forme résolue de la contrainte $\alpha = \langle F\,\gamma\rangle = \langle G\,\gamma_1\rangle = \langle G\,\gamma_2\rangle \wedge \gamma_1 = \langle \alpha \to \alpha\rangle \wedge \gamma_2 = \langle \beta \to \alpha\rangle \wedge \beta =$

$\langle F\,\gamma = G\,\gamma\rangle$: il suffit de décomposer $G\,\gamma_1 = G\,\gamma_2$ qui va permettre d'obtenir après fusions des multi-équations sur γ_1 et γ_2 et décompositions de $\rightarrow$ que $\alpha = \beta$.

Pour limiter le nombre d'expansions, la stratégie consiste à appliquer agressivement les autres règles jusqu'à ce qu'elles ne sont plus applicables. La seule règle applicable pour avancer est alors l'expansion. Informellement, on s'assure ainsi d'avoir collecter toutes les informations possibles via fusion et décomposition. La stratégie est construite autour de trois groupes de règles composées qui combinent plusieurs règles élémentaires de façon ordonnée (1) les règles de génération de contraintes sont combinées (5.2.1) avec les règles Cut pour maintenir les égalités sur des petits termes et PDispatch et NPDispatch pour répartir les termes des égalités dans les vues et ne pas avoir de termes non traités ; les contraintes résultantes respectent des invariants énoncées en 5.2.2 ; (2) la règle Merge-Decompose (5.2.3) compose la règle Merge avec autant de décompositions (Decompose) que possible entre les deux multi-équations fusionnées et autant de retraits de doublons (NPClean et PClean) que possibles sur les vues décomposées. La seule règle de résolution élémentaire applicable sur la contrainte ainsi obtenue est l'expansion ; (3) Expanse-Decompose (5.2.4) est utilisée uniquement lors de sa résolution de multi-équations, et combine Expanse, Cut, PCut, PDispatch, NPDispatch, Decomposeet PClean pour maintenir les invariants (5.2.2).

Pour définir les règles ci-dessous, on utilise la notation $\longrightarrow^{\infty}$ pour dénoter l'application d'une règle jusqu'à ce que ce ne soit plus possible. Comme on a montré la terminaison des règles au dessus, cela correspond à un nombre fini d'applications. On note $\longrightarrow^{?}$ l'application d'une règle au plus une fois.

5.2.1 Génération de contraintes en petits termes et avec vues

Les règles de génération de contraintes sont redéfinies pour faire en sorte qu'elles placent les termes construits des égalités qu'elles introduisent dans des vues (NPDispatchet PDispatch) et pour qu'elles soient écrites en petits termes (Cut).

Les règles initiales (3.2.2) nécessitent peu de changements : elles sont déjà en petits termes. Seules les règles Abs, App et Pair doivent être modifées. Il suffit juste d'appliquer une fois PDispatch pour mettre les termes construits dans des vues productives. Les nouvelles règles sont :

$$\llbracket(\lambda x.\,e) : \alpha\rrbracket \;\longrightarrow\; \exists\alpha_1\exists\alpha_2.\,(\mathsf{def}\,x = \alpha_1\,\mathsf{in}\,\llbracket e : \alpha_2\rrbracket \wedge \alpha = \langle\alpha_1 \rightarrow \alpha_2\rangle) \qquad \textbf{Abs'}$$

$$\llbracket e_1\,e_2 : \alpha\rrbracket \;\longrightarrow\; \exists\beta_1\exists\beta_2.\,(\llbracket e_1 : \beta_2\rrbracket \wedge \beta_2 = \langle\beta_1 \rightarrow \alpha\rangle \wedge \llbracket e_2 : \beta_1\rrbracket) \qquad \textbf{App'}$$

$$\llbracket(e_1, e_2) : \alpha\rrbracket \;\longrightarrow\; \exists\alpha_1\exists\alpha_2.\,(\llbracket e_1 : \alpha_1\rrbracket \wedge \llbracket e_2 : \alpha_2\rrbracket \wedge \alpha = \langle\alpha_1 * \alpha_2\rangle) \qquad \textbf{Pair'}$$

La règle Annot introduit l'égalité $\alpha = \tau_0$ qu'il faut découper en petits termes et placer dans les vues. Pour cela, on définit la règle Cut-Dispatch qui applique la règle Cut jusqu'à ce que les multi-équations soient en petits termes, suivie des règles PDispatch et NPDispatch qui mettent tous les types construits dans des vues. La nouvelle règle Annot'

est donc la composition de la règle ANNOT suivie de la règle CUT-DISPATCH appliquée au plus une fois sur l'égalité $\alpha = \tau_0$ [5].

$$\epsilon \xrightarrow{\text{Cut-Dispatch}} \mathcal{C} \triangleq \epsilon \left(\xrightarrow{\text{Cut}}\right)^{\infty} \mathcal{C}_1 \left(\xrightarrow{\text{NPDispatch}}\right)^{\infty} \mathcal{C}_2 \left(\xrightarrow{\text{PDispatch}}\right)^{\infty} \mathcal{C}$$

$$\mathcal{C}_1 \xrightarrow{\text{Annot'}} \mathcal{C}_2 \triangleq \mathcal{C}_1 \xrightarrow{\text{Annot}} \mathcal{C}_3 \left(\xrightarrow{\text{Cut-Dispatch}}\right)^{?} \mathcal{C}_2 \qquad \textbf{Annot'}$$

Les vues productives produites sont toutes des singletons et comme aucun MERGE n'a été réalisé, les multi-équations obtenus à l'issue de l'utilisation de ces règles sont toutes de la forme $\alpha = \langle F\,\vec{\beta}\rangle$ ou $\alpha = [F\,\vec{\beta}] = \beta_i$.

5.2.2 Invariants

Le nouveau jeu de règles de génération de contraintes est composée de VAR, ABS', APP', PAIR', LET, NEW et ANNOT' [6] et les règles d'unifications sont EXAND, STUTTER, SINGLE, TRUE, CYCLE, FAIL [7] auxquelles s'ajoutent les règles définis ci-dessous MERGE-DECOMPOSE et EXPANSE-DECOMPOSE.

Toutes ces règles ne font que combiner les règles élémentaires précédentes (4.3.3), les invariants décrits précédemment (4.3.4) sont toujours vrais et on a en plus de nouveaux invariants :

Invariant 6. *Dans une contrainte, chaque multi-équation est telle que*

(a) Tous les types construits sont dans une vue ;

(b) Tous les types construits sont en petits termes ;

(c) Deux vues productives n'ont aucun constructeur de type en commun ;

(d) Une vue productive a un unique représentant.

Démonstration. Les règles de génération génèrent des contraintes qui respectent ces invariants. (a) PDISPATCH et NPDISPATCH sont appliquées sur tous nouveaux termes construits ; (b) CUT est appliquée autant que possible ; (c) et (d) les vues productives sont des singletons et les multi-équations ont au plus une vue. On montre dans la suite que les nouvelles règles MERGE-DECOMPOSE (5.2.3) et EXPANSE-DECOMPOSE (5.2.4) préservent ces invariants. □

5. Si $\tau_0 \in \mathcal{V}$, CUT-DISPATCH n'est pas appliquée.

6. Les règles sans ' sont les anciennes règles non modifiées, définis en 3.2 et les autres sont définies ci-dessus.

7. Les règles d'unification intouchées définies en 4.2.2

5.2.3 Fusion et décompositions agressives

L'objectif ici est de définir une règle de fusion de multi-équations qui maintienne ces invariants.

Decompose-Clean

$$\epsilon \xrightarrow{\textsc{Decompose-Clean}} \mathcal{C} \ \triangleq\ \epsilon \left(\xrightarrow{\textsc{Clash}} \cup \xrightarrow{\textsc{Decompose}}\right) \mathcal{C}_1 \left(\xrightarrow{\textsc{PClean}}\right)^{\infty} \mathcal{C}$$

Merge-Decompose

$$\epsilon_1 \wedge \epsilon_2 \xrightarrow{\textsc{Merge-Decompose}} \mathcal{C} \ \triangleq\ \epsilon_1 \wedge \epsilon_2 \xrightarrow{\textsc{Merge}} \epsilon_3 \left(\xrightarrow{\textsc{NPClean}}\right)^{\infty} \epsilon_4 \left(\xrightarrow{\textsc{Decompose-Clean}}\right)^{\infty} \mathcal{C}$$

La règle Decompose-Clean est une règle de décomposition qui retire tous les doublons. Par exemple, $\langle F_0\,\alpha = F_1\,\alpha = F_2\,\alpha\rangle = \langle F_0\,\beta = F_1\,\beta\rangle = \epsilon$ devient après application de Decompose-Clean$\langle F_0\,\alpha = F_1\,\alpha = F_2\,\alpha\rangle = \epsilon \wedge \alpha = \beta$.

La règle Merge-Decompose fusionne deux multi-équations partageant une variable de type, nettoie la vue non-productive et réalisent toutes les décompositions possibles. Cette règle (ainsi que les règles structurelles nécessaires) doit être appliquée autant de fois que nécessaire avant d'appliquer la règle d'expansion définie ci-dessous. Elle préserve les invariants (a) et (b) car les nouvelles égalités générées par la décomposition ne contiennent que des variables de type. (c) est préservé car il n'est plus possible de faire de décomposition. (d) est préservé par application agressive de la règle PClean.

5.2.4 Expansion

La résolution des multi-équations nécessite l'utilisation de la règle Expanse. On redéfinit ici la règle pour qu'elle préserve les invariants et s'applique efficacement.

Expanse-Decompose

$$\epsilon \xrightarrow[(1)]{\textsc{Expanse-Decompose}} \mathcal{C} \ \triangleq\ \epsilon \xrightarrow[(1)]{\textsc{Expanse}} \mathcal{C}_1 \left(\xrightarrow{\textsc{PCut}}\right)^{\infty} \mathcal{C}_2 \left(\xrightarrow{\textsc{Cut-Dispatch}}\right)^{\infty} \mathcal{C}_3 \left(\xrightarrow{\textsc{Decompose-Clean}}\right)^{?} \mathcal{C}$$

où la condition d'application (i) la vue expansée ν est une de celles de plus haut rang dans ϵ ; (ii) le terme expansé est le représentant de la vue ν.

Pour clarifier la règle, on indique que pour ϵ de la forme $(\langle\nu_1\rangle = \langle\nu_2\rangle)$, on a :

— $\mathcal{C}_1$ de la forme $\exists\vec{\alpha}.\,(\langle G\,\vec{\tau} = \nu_1\rangle = \langle\nu_2\rangle = \epsilon_1 \bigwedge_{i\in(\!|F|\!)} \alpha_i = \beta_i)$

— $\mathcal{C}_2$ de la forme $\exists\vec{\alpha}.\,\exists\vec{\gamma},\,(\langle G\,\vec{\gamma} = \nu_1\rangle = \langle\nu_2\rangle = \epsilon_1 \wedge \bigwedge_{i\in(\!|F|\!)} \alpha_i = \beta_i \wedge \vec{\gamma} = \vec{\tau})$

La condition (i) est là pour faire moins d'expansions. Les invariants nous indiquent que chaque vue contient un seul morceau d'arbre (invariant 5) et que dans une multi-équation, les morceaux d'arbres sont tous disjoints (invariant 6(c)). Le but de la règle Expanse est de prolonger le morceau d'arbre d'une vue, via une expansion, pour qu'il en rejoigne un

autre et qu'une décomposition soit possible. Comme l'extension se fait vers le bas, il faut choisir la vue de plus haut grand pour s'assurer de faire une expansion utile, c'est-à-dire la vue dont la racine du morceau d'arbre (i.e. le représentant de la vue) est la plus haute. Enfin, la condition (ii) est une application de l'invariant 5 : le seul type d'une vue qu'il est utile d'expanser est son représentant puisque dans la contrainte, tous les autres types de la vue s'expansent en tête vers le représentant.

Enfin, on notera qu'il ne peut y avoir qu'au plus une décomposition après avoir expanser. En effet, comme toutes les vues sont disjointes, (invariant 6(c)), le nouveau constructeur de tête G introduit par l'expansion apparaît au plus dans une vue productive de la multi-équation, avec laquelle il est alors possible de faire une décomposition.

Expanse-Decompose préservent les invariants (a)- et (b) par application de Cut-Dispatch et PCut. (c) est préservé car seule la règle élémentaire Expanse introduit une éventuelle décomposition, et elle est prise en charge par le Decompose-Clean. (d) est préservé par applications répétées de la règle PClean.

5.2.5 Stratégie

La résolution d'une contrainte $\mathsf{let}\, x = \mathcal{C}_1 \,\mathsf{in}\, \mathcal{C}_2$ est équivalente à substituer la variable de programme x par la contrainte $\mathcal{C}_1$ dans $\mathcal{C}_2$. Comme la contrainte $\mathcal{C}_1$ est copiée, elle doit être résolue avant la contrainte let pour ne pas avoir à faire plusieurs fois sa résolution. La résolution de multi-équation prend donc place avant de sortir de la partie gauche d'une contrainte let. La stratégie de résolution consiste à appliquer la règle Merge-Decompose autant que possible pour effectuer toutes les fusions de multi-équations et de vues possibles. Ensuite seulement, la règle Expanse-Decompose est appliquée pour amener les multi-équations dans leur forme résolue.

$$\mathcal{C}_1 \xrightarrow{\textsc{Resolve}} \mathcal{C}_2 \triangleq \mathcal{C}_1 \left(\xrightarrow{\textsc{Merge-Decompose}}\right)^\infty \mathcal{C}_3 \left(\xrightarrow{\textsc{Expanse-Decompose}} \left(\xrightarrow{\textsc{ExAnd}}\right)^\infty \left(\xrightarrow{\textsc{Merge-Decompose}}\right)^\infty\right)^\infty \mathcal{C}_2$$

Notre stratégie préserve les solutions car elle n'utilise que les règles élémentaires définies précedemment qui préservent les solutions.

À l'issue de l'application de notre stratégie, il faut ajouter autant d'applications de la règle Stutter qui retirent les variables en double dans une même multi-équation pour obtenir des multi-équations en forme résolue : aucune des règles élémentaires ne peut alors être appliquées. Les invariants (a), (b) et (c) assurent en effet que les règles NPDispatch, PDispatch, PCut, PCut, Decompose et Clash ne sont pas applicables. Enfin, comme les règles Merge-Decompose et Expanse-Decompose ne sont plus applicables, alors les règles Merge et Expanse ne le sont pas non plus.

5.3 Limites de la stratégie

Notre stratégie réduit les expansions réalisées en les repoussant autant que possible et la mémorisation des expansions en est un élément clé. Cependant, c'est aussi cependant un

élément potentiellent coûteux en mémoire.

De plus, notre stratégie n'est pas optimum : le choix de la multi-équation dans laquelle expansée reste arbitraire lorsque que plusieurs sont possibles et peut induire plus ou moins de décompositions et donc changer le nombre d'expansions à réaliser pour atteindre une forme normale (résultant en des formes normales différentes). Nous n'avons cependant pas trouver de critère simple et non arbitraire pour choisir une multi-équation plutôt qu'une autre.

6 Élaboration

Dans l'inférence de types, le solveur de contraintes a pour objectif de déterminer si un programme est bien typé : il retourne donc vrai ou faux en fonction de la satisfaisabilité de la contrainte. Ensuite, l'élaboration est en charge de construire la représentation explicitement typée du programme. C'est une étape qui, comme la génération de contraintes, dépend du langage source et nécessite de parcourir le programme. Dans [8], Pottier propose une implémentation de l'inférence de types avec contraintes qui réalise simultanément ces deux étapes (6.1). Un intérêt essentiel de cette méthode est de faciliter la maintenance du typeur lors d'ajouts d'extensions du langage. Notons par ailleurs que la résolution de contraintes n'est pas un processus local : dans la contrainte $\exists\alpha.\ \mathcal{C}_1 \wedge \mathcal{C}_2$ la valeur finale d'une variable de type ne peut être déterminée qu'une fois $\mathcal{C}_2$ résolue. Pour pallier ce problème, Inferno utilise des continuations pour réaliser l'élaboration qui sont construites en même temps que la contrainte et qui ne sont appliquées qu'une fois que la résolution de celle-ci est achevée.

Les abréviations change l'élaboration en cela qu'un même type peut s'exprimer de plusieurs façons différentes. Il y a donc plusieurs stratégies possibles pour l'élaborer. Par exemple, dans

```
type foo = int * int
type 'a t = 'a -> foo
let f : int t = fun x -> (x, x)
let r  = f 1
```

r est de type $\mathsf{int} \to \mathsf{int}$ mais aussi de type foo. Dans le premier cas, le témoin choisi est le type le plus expansé, alors que dans le second cas, on garde la forme non expansée. On discute dans la suite des différentes stratégies d'élaboration possibles 6.2.

6.1 Contraintes à valeurs

Pour permettre l'élaboration, les contraintes ne retournent pas uniquement une information de satisfaisabilité mais aussi des *valeurs*, déterminées par le solveur de contraintes (et affectées par effet de bord). Ainsi par exemple[8], la contrainte $\mathcal{C}_1 \wedge \mathcal{C}_2$ retourne une paire de valeur (V_1, V_2) composée des valeurs retournées respectivement par $\mathcal{C}_1$ et $\mathcal{C}_2$, une

8. Cf [8] pour tous les cas, notamment le cas let qui présente des subtilités.

contrainte égalité $\tau_1 = \tau_2$ retourne la valeur unit et $\exists\alpha.\ \mathcal{C}$ produit une paire (T, V) où T est un type du langage destination satisfaisant l'existentiel et V la valeur retournée par $\mathcal{C}$.

Ainsi, le processus d'inférence pour le cas de l'expression $(\lambda x.\, e)$ est le suivant. La génération de contraintes lui associe la variable de type à inférer α et produit la contrainte $\exists\alpha_1\exists\alpha_2.\ (\mathsf{def}\, x = \alpha_1 \,\mathsf{in}\, [\![e : \alpha_2]\!] \wedge \alpha = \alpha_1 \rightarrow \alpha_2)$. À l'issue de la résolution de contraintes, α est associée à un descripteur, qui se traduit en un type T dans le langage source et la contrainte $[\![e : \alpha_2]\!]$ retourne une valeur V. Le terme élaboré est alors $\lambda x : T.\, V$. L'article [8] détaille les autres cas et l'implémentation faite.

6.2 Stratégies d'élaboration

Avec les abréviations, le descripteur d'une variable de type n'est plus soit rien soit un unique type mais soit (1) un ensemble potentiellement vide de types non productifs soit (2) une paire composée de deux ensembles de types équivalents : des types productifs et des types non productifs. La valeur produite par une contrainte existentielle n'est donc plus un unique témoin mais un ensemble possible de témoins tous sémantiquement équivalents mais différents syntaxiquement. La stratégie d'élaboration consiste alors à choisir un sous-ensemble de témoins. Il n'est pas nécessaire d'appliquer une stratégie dans le cas (1) : la variable de type que l'on souhaite élaborer est généralisable. On peut cependant choisir de l'afficher comme un type non producif. Par contre, si le descripteur est une paire, i.e. s'il y a au moins un témoin productif possible, il faut appliquer une stratégie d'élaboration pour déterminer les témoins et au moins un témoin doit être retourné.

On note Φ une stratégie d'élaboration des abréviations. Elle associe à un descripteur l'ensemble des témoins choisis. Plusieurs stratégies peuvent être utilisées en faisant une union des résultats ou par composition de stratégies. La stratégie par défaut, Φ_{all} retourne toutes les termes du descripteur. L'idée est de proposer plusieurs stratégies, de sorte à adapter l'élaboration en fonction des circonstances ou des choix de l'utilisateur. Par exemple, dans le programme $\mathsf{let\ f} = \mathsf{new}\,\alpha\,\mathsf{in}\,(\mathsf{fun}\ x \rightarrow 1) : F\,\alpha$ où $F\,\alpha \cong \alpha \rightarrow \alpha$, une élaboration vers un terme explicitement typé pourrait être $\mathsf{let\ f} : \mathsf{int} \rightarrow \mathsf{int} = (\mathsf{fun}\ (x : \mathsf{int}) \rightarrow 1) : \mathsf{int}\,F$. La stratégie d'élaboration pour l'annotation est alors différente de celle utilisée pour le type de f.

On permet des stratégies qui retournent plusieurs témoins puisqu'il est tout à fait possible d'avoir un affichage des types donnant les égalités pertinentes. Ocaml le fait déjà via le module emacs Merlin [9]. Un autre intérêt de l'affichage de plusieurs témoins est de limiter l'arbitraire, comme on trouve en Ocaml :

```
type 'a t = int -> 'a and 'a u = 'a -> bool
let f : bool t = fun x -> true and  g : int u = fun x -> false
let r = if true then f else g (* val r : bool t = <fun> *)
```

9. Avec Merlin, il est possible de demander 2 fois consécutives le type d'un terme. Le second affichage donne la forme totalement expansée.

```
let s = if true then g else f (* val s : int u = <fun> *)
```

Les types issus de la vue productive forment un sous-arbre continu dans la forêt décrite par les définitions d'abreviations (cf invariant 5). Une vue productive a donc des feuilles et une racine unique : son représentant (cf invariant 6). On propose donc trois stratégies sur ces termes : Φ_{pall} retourne tous les termes productifs, Φ_{proot} retourne le représentant de la vue et Φ_{pleaves} retourne les feuilles. La stratégie Φ_{proot} choisit un témoin unique qui est le type le plus expansé. Elle met en avant les égalités entre types en affichant de la même façon tous les types égalisés. Au contraire, la stratégie Φ_{pleaves} peut retourner plusieurs témoins qui sont les termes les moins expansés. C'est une stratégie qui cible la concision qui est très importante pour les objets. L'ensemble des types issus de la vue non productive ne bénéficie pas des mêmes invariants sur les rangs que la vue productive puisqu'on a mis en avant l'efficacité pour les règles de résolutions pour les abréviations non productives (NPDispatch et NPClean). Ainsi, il n'y a pas de représentant dans une vue non-productive. En conséquent, on propose des stratégies très simples : Φ_{npall} et $\Phi_{\text{npnothing}}$ qui respectivement retournent tous les termes non productifs et filtrent tout les termes non productifs.

7 Conclusion

Faute de place, on réfère le lecteur à la fiche de synthèse () qui résume le travail réalisé et ses perspectives.

A Références

[1] Jörgen Gustavsson and Josef Svenningsson. Constraint abstractions. pages 63–83, 01 2001.

[2] Bastiaan Heeren, Jurriaan Hage, and S Swierstra. Generalizing hindley-milner type inference algorithms. 08 2002.

[3] R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. Amer. Math. Soc*, 146 :29–60, December 1969.

[4] Gérard Huet. *Résolution d'équations dans les langages d'ordre 1, 2, ..., ω*. PhD thesis, Université de Paris 7, Paris, France, 1976.

[5] J. JOUANNAUD. Solving equations in abstract algebras : A rule-based survey of unification. *Computational Logic*, pages 257–321, 1991.

[6] Alberto Martelli and Ugo Montanari. An Efficient Unification Algorithm. *Transactions on Programming Languages and Systems (TOPLAS)*, 4(2) :258–282, 1982.

[7] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17 :348–375, 1978.

[8] François Pottier. Hindley-Milner Elaboration in Applicative Style. In *ICFP 2014 : 19th ACM SIGPLAN International Conference on Functional Programming*, Goteborg, Sweden, September 2014. ACM.

[9] François Pottier and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

[10] Didier Rémy and Jérôme Vouillon. Objective ML : An effective object-oriented extension to ML. *Theory And Practice of Object Systems*, 4(1) :27–50, 1998. A preliminary version appeared in the proceedings of the 24th ACM Conference on Principles of Programming Languages, 1997.

[11] J. A. Robinson. Computational logic : The unification computation. *Machine intelligence*, 6(63-72) :10–1, 1971.

[12] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2) :215–225, April 1975.

[13] Mitchell Wand. A simple algorithm and proof for type inference. *Fundamenta Infomaticae*, 10 :115–122, 1987.

B Table des matières

1 Langage source **3**
- 1.1 Grammaire . . . 3
- 1.2 Syntaxe des types . . . 3
- 1.3 Types clos . . . 4

2 Sur les abréviations **4**
- 2.1 Expansion . . . 4
- 2.2 Abréviations non productives . . . 5
- 2.3 Représentation des abréviations . . . 5
- 2.4 Décomposition et paramètres fantômes . . . 5
- 2.5 Définitions valides et abréviations récursives . . . 6

3 Génération de contraintes **8**
- 3.1 Syntaxe du langage des contraintes . . . 8
- 3.2 Génération de contraintes . . . 9

4 Algorithme d'unification **10**
- 4.1 Interprétation et satisfaisibilité des contraintes . . . 10
- 4.2 Algorithme d'unification sans abréviation . . . 10
- 4.3 Algorithme d'unification avec abréviations . . . 12

5 Stratégie d'unification **15**
- 5.1 Rang et représentant . . . 15
- 5.2 La stratégie . . . 16
- 5.3 Limites de la stratégie . . . 19

6 Élaboration **19**
- 6.1 Contraintes à valeurs . . . 19
- 6.2 Stratégies d'élaboration . . . 20

7 Conclusion **20**

A Références **21**

B Table des matières **22**

C Exemple **23**

D Preuves **24**

C Exemple

Voici un exemple qui montre le fonctionnement de la génération de contrainte et l'algorithme d'unification avec quelques modifications pour rendre son exécution lisible : certaines règles sont utilisés systèmatiquement : les existentiels sont automatiquement sortis (règle ExAnd), les variables en double dans une multi-équations sont retirées (Stutter). On applique notre algorithme sur le terme $\lambda(x : K\alpha).\, f\, x$ avec f de type $G\, \mathsf{int}$ et les définitions d'abréviations suivantes :

$$
\begin{aligned}
K\,\alpha &\cong \alpha * \alpha \\
F\,\alpha &\cong \alpha \rightarrow \alpha \\
G\,\alpha &\cong F\,(K\,\alpha)
\end{aligned}
$$

Génération de contraintes

$[\![\lambda(x : K\,\alpha).\, f x \;: \alpha_0]\!]$

$\xrightarrow{\text{Abs}} \exists \alpha_1, \alpha_2.\,(\mathsf{def}\, x = \alpha_1 \,\mathsf{in}\, [\![f\,x : \alpha_2]\!] \wedge \alpha_0 = \alpha_1 \to \alpha_2 \wedge \alpha_1 = K\,\alpha)$

$\xrightarrow{\text{App}} \exists \alpha_1, \alpha_2, \alpha_3, \alpha_4.\,(\mathsf{def}\, x = \alpha_1 \,\mathsf{in}\, [\![f : \alpha_4]\!] \wedge [\![x : \alpha_3]\!] \wedge \alpha_4 = \alpha_3 \to \alpha_2 \wedge \alpha_0 = \alpha_1 \to \alpha_2 \wedge \alpha_1 = K\,\alpha)$

$\xrightarrow{f : G\,\mathsf{int}} \exists \alpha_1, \alpha_2, \alpha_3, \alpha_4.\,(\mathsf{def}\, x = \alpha_1 \,\mathsf{in}\, \alpha_4 = G\,\mathsf{int} \wedge [\![x : \alpha_3]\!] \wedge \alpha_4 = \alpha_3 \to \alpha_2 \wedge \alpha_0 = \alpha_1 \to \alpha_2 \wedge \alpha_1 = K\,\alpha)$

$\xrightarrow{\text{Var}} \exists \alpha_1, \alpha_2, \alpha_3, \alpha_4.\,(\mathsf{def}\, x = \alpha_1 \,\mathsf{in}\, \alpha_4 = G\,\mathsf{int} \wedge x\,\alpha_3 \wedge \alpha_4 = \alpha_3 \to \alpha_2 \wedge \alpha_0 = \alpha_1 \to \alpha_2 \wedge \alpha_1 = K\,\alpha)$

Unification

$\xrightarrow{x \leftarrow \lambda\alpha.\,\alpha = \alpha_1} \exists \alpha_1, \alpha_2, \alpha_3, \alpha_4.\,(\alpha_4 = G\,\mathsf{int} \wedge \alpha_1 = \alpha_3 \wedge \alpha_4 = \alpha_3 \to \alpha_2 \wedge \alpha_0 = \alpha_1 \to \alpha_2 \wedge \alpha_1 = K\,\alpha)$

$\xrightarrow{\text{Merge}} \exists \alpha_1, \alpha_2, \alpha_3, \alpha_4.\,(\alpha_4 = G\,\mathsf{int} \wedge \alpha_4 = \alpha_3 \to \alpha_2 \wedge \alpha_0 = \alpha_1 \to \alpha_2 \wedge \alpha_1 = \alpha_3 = K\,\alpha)$

$\xrightarrow{\text{Merge}} \exists \alpha_1, \alpha_2, \alpha_3, \alpha_4.\,(\alpha_4 = G\,\mathsf{int} = \alpha_3 \to \alpha_2 \wedge \alpha_0 = \alpha_1 \to \alpha_2 \wedge \alpha_1 = \alpha_3 = K\,\alpha)$

$(\xrightarrow{\text{PDispatch}})^4\, \exists \alpha_1, \alpha_2, \alpha_3, \alpha_4.\,(\alpha_4 = \langle G\,\mathsf{int}\rangle = \langle \alpha_3 \to \alpha_2\rangle \wedge \alpha_0 = \langle \alpha_1 \to \alpha_2\rangle \wedge \alpha_1 = \alpha_3 = \langle K\,\alpha\rangle)$

$\xrightarrow{\text{PCut}}\xrightarrow{\text{PDispatch}} \exists \alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5.\,(\alpha_4 = \langle G\,\alpha_5\rangle = \langle \alpha_3 \to \alpha_2\rangle \wedge \alpha_5 = \langle \mathsf{int}\rangle$
$\quad \wedge\, \alpha_0 = \langle \alpha_1 \to \alpha_2\rangle \wedge \alpha_1 = \alpha_3 = \langle K\,\alpha\rangle)$

$\xrightarrow{\text{Expanse}} \exists \alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5.\,(\alpha_4 = \langle G\,\alpha_5 = F\,(K\,\alpha_5)\rangle = \langle \alpha_3 \to \alpha_2\rangle \wedge \alpha_5 = \langle \mathsf{int}\rangle$
$\quad \wedge\, \alpha_0 = \langle \alpha_1 \to \alpha_2\rangle \wedge \alpha_1 = \alpha_3 = \langle K\,\alpha\rangle)$

$\xrightarrow{\text{PCut}}\xrightarrow{\text{PDispatch}} \exists \alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6.\,(\alpha_4 = \langle G\,\alpha_5 = F\,\alpha_6\rangle = \langle \alpha_3 \to \alpha_2\rangle \wedge \alpha_6 = \langle K\,\alpha_5\rangle \wedge \alpha_5 = \langle \mathsf{int}\rangle$
$\quad \wedge\, \alpha_0 = \langle \alpha_1 \to \alpha_2\rangle \wedge \alpha_1 = \alpha_3 = \langle K\,\alpha\rangle)$

$\xrightarrow{\text{Expanse}} \exists \alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6.\,(\alpha_4 = \langle G\,\alpha_5 = F\,\alpha_6 = \alpha_6 \to \alpha_6\rangle = \langle \alpha_3 \to \alpha_2\rangle \wedge \alpha_6 = \langle K\,\alpha_5\rangle \wedge \alpha_5 = \langle \mathsf{int}\rangle$
$\quad \wedge\, \alpha_0 = \langle \alpha_1 \to \alpha_2\rangle \wedge \alpha_1 = \alpha_3 = \langle K\,\alpha\rangle)$

$\xrightarrow{\text{Decompose}}\xrightarrow{\text{PClean}} \exists \alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6.\,(\alpha_4 = \langle G\,\alpha_5 = F\,\alpha_6 = \alpha_6 \to \alpha_6\rangle$
$\quad \wedge\, \alpha_6 = \alpha_3 \wedge \alpha_6 = \alpha_2 \wedge \alpha_6 = \langle K\,\alpha_5\rangle \wedge \alpha_5 = \langle \mathsf{int}\rangle \wedge \alpha_0 = \langle \alpha_1 \to \alpha_2\rangle \wedge \alpha_1 = \alpha_3 = \langle K\,\alpha\rangle)$

$(\xrightarrow{\text{Merge}})^4\, \exists \alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6.\,(\alpha_4 = \langle G\,\alpha_5 = F\,\alpha_6 = \alpha_6 \to \alpha_6\rangle$
$\quad \wedge\, \alpha_6 = \alpha_3 = \alpha_2 = \alpha_1 = \langle K\,\alpha_5\rangle = \langle K\,\alpha\rangle) \wedge \alpha_5 = \langle \mathsf{int}\rangle \wedge \alpha_0 = \langle \alpha_1 \to \alpha_2\rangle$

$\xrightarrow{\text{Decompose}}\xrightarrow{\text{PClean}} \exists \alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6.\,(\alpha_4 = \langle G\,\alpha_5 = F\,\alpha_6 = \alpha_6 \to \alpha_6\rangle$
$\quad \wedge\, \alpha_6 = \alpha_3 = \alpha_2 = \alpha_1 = \langle K\,\alpha_5\rangle \wedge \alpha_5 = \alpha \wedge \alpha_5 = \langle \mathsf{int}\rangle \wedge \alpha_0 = \langle \alpha_1 \to \alpha_2\rangle)$

$\xrightarrow{\text{Merge}} \exists \alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6.\,(\alpha_4 = \langle G\,\alpha_5 = F\,\alpha_6 = \alpha_6 \to \alpha_6\rangle$
$\quad \wedge\, \alpha_6 = \alpha_3 = \alpha_2 = \alpha_1 = \langle K\,\alpha_5\rangle \wedge \alpha_5 = \alpha = \langle \mathsf{int}\rangle \wedge \alpha_0 = \langle \alpha_1 \to \alpha_2\rangle)$

On obtient que le type de l'expression $\lambda(x : K\,\alpha).\, f\,x$ est $K\,\mathsf{int} \to K\,\mathsf{int}$.

D Preuves

Pour les preuves ci-dessous, pour un ensemble de termes, on notera $\phi\nu$ pour $\phi\tau_\nu$ où $\tau_\nu \in \nu$.

Invariant 1 sur les vues non productives

Démonstration. La preuve est faite par cas sur les règles de réécriture. Les règles de génération de contraintes génèrent des multi-équations avec des vues non productives vides, qui respectent donc l'invariant. Pour les règles de résolution, on a besoin de considérer seulement les règles qui altèrent la vue non-productive, c'est-à-dire les règles MERGE, NPDISPATCH et NPCLEAN.

- comme l'invariant est vrai avant application des règles, la règle MERGE fait l'union de deux vues bien formées. La vue résultante est bien formée.
- la règle NPCLEAN retire un élément de la vue non productive et préserve donc l'invariant.
- la règle NPDISPATCH ajoute le terme $F\,\vec{\tau}$ dans la vue non productive qui, par hypothèse d'application de la règle, respecte l'invariant.

□

Invariant 2 sur les vues non productives

Démonstration. Le sens direct $([\delta] = \epsilon) \Rrightarrow (\epsilon)$ est trivialement toujours vrai.

On travaille par cas sur les règles de réécriture. Les règles de génération de contraintes créé des vues non productives vides donc préservent l'invariant. On considère les trois règles de résolution qui affectent ces vues : MERGE, NPDISPATCH et NPCLEAN.

Pour la règle MERGE, soit ϕ une solution de la contrainte $\alpha = \epsilon_1 = \epsilon_2$. ϕ est aussi solution de $\alpha = \epsilon_1$ et de $\alpha = \epsilon_2$. Comme l'invariant est vrai avant application de la règle, ϕ est aussi solution de $\alpha = [\delta_1] = \epsilon_1$ et de $\alpha = [\delta_2] = \epsilon_2$ donc $\phi\,\alpha =_E \phi\,\delta_1 =_E \phi\,\epsilon_1$ et $\phi\,\alpha =_E \phi\,\delta_2 =_E \phi\,\epsilon_2$ et par transitivité, $\phi\,\alpha =_E \phi\,\delta_1 =_E \phi\,\epsilon_1 =_E \phi\,\delta_2 =_E \phi\,\epsilon_2$ donc ϕ est solution de $\alpha = [\delta_1 = \delta_2] = \epsilon_1 = \epsilon_2$

Pour la règle NPCLEAN, soit ϕ une solution de ϵ. Comme l'invariant est vrai avant l'application de la règle, ϕ est aussi une solution de $[F\,\vec{\tau}_0 = F\,\vec{\tau}_1 = \delta] = \epsilon$ et est donc aussi solution de $[F\,\vec{\tau}_0 = \delta] = \epsilon$ qui est moins contraint.

Pour la règle NPDISPATCH, soit ϕ solution de $\tau_i = \epsilon$. On a donc $\phi\,\epsilon =_E \phi\,\tau_i$. Comme $F\,\vec{\tau}$ s'expanse en tête vers τ_i, on a $\phi\tau_i = \phi(F\,\vec{\tau})$ et donc ϕ est solution de $F\,\vec{\tau} = \tau_i = \epsilon$ (1). Comme l'invariant est vrai avant application de la règle, ϕ est aussi solution de $[\delta = F\,\vec{\tau}] = \epsilon$, i.e. de $\delta = F\,\vec{\tau} = \epsilon$ (2). En combinant (1) et (2), on obtient que ϕ est solution de $\delta = F\,\vec{\tau} = \tau_i = \epsilon$. Comme $[\delta = F\,\vec{\tau}]$ est bien formé, ϕ est aussi solution de $[\delta = F\,\vec{\tau}] = \tau_i = \epsilon$. □

Invariant 3 sur les vues productives

Démonstration. Par cas sur les règles de réécriture. Les règles de génération de contraintes générent des multi-équations sans vues productives donc respectant l'invariant. Dans les règles de résolution, seules les règles PDISPATCH, PCLEAN et EXPANSE modifient les vues productives :

- la règle PDISPATCH créé une nouvelle vue productive singleton, qui, par hypothèse d'application, respecte l'invariant.
- PCLEAN retire un élément de la vue et ne créé jamais une vue vide.
- Comme l'invariant est vrai avant application de la règle, DECOMPOSE fait l'union de deux vues qui respectent l'invariant.
- EXPANSE ajoute l'expansion en tête d'un type productif dans la vue. Par définition d'un type productif, son expansion est productive.

□

Invariant 4 sur les vues productives

Démonstration. On travaille par cas sur les règles de réécriture. Les règles de génération de contraintes ne générent pas de vues productives. Dans les règles de résolution, seules les règles PCLEAN, PDISPATCH, DECOMPOSE et EXPANSE modifient les vues productives.

— La règle PDISPATCH créé une nouvelle vue productive singleton. La règle PCLEAN préserve l'invariant car tous les sous-ensembles de $F\,\vec{\alpha} = \nu$ sont des sous ensembles de $F\,\vec{\alpha} = F\,\vec{\beta} = \nu$.

— Pour la règle DECOMPOSE, soit $\nu_0 \subseteq (F\,\vec{\alpha} = F\,\vec{\beta} = \nu = \nu')$. Écrivons ν_0 sous la forme $(\nu_1 = \nu_2)$ telle que $\nu_1 \subseteq (F\,\vec{\alpha} = \nu)$, $\nu_2 \subseteq (F\,\vec{\beta} = \nu')$. On a deux cas : (a) ν_1 et ν_2 sont non vides, et (b) un des deux est vides. Dans le cas (a) on utilise le fait que l'invariant soit vrai avant application de la règle : $\langle\nu_1\rangle = \langle\nu_2\rangle = \epsilon \Rrightarrow \langle F\,\vec{\alpha} = \nu\rangle = \langle\nu_2\rangle = \epsilon \Rrightarrow \langle F\,\vec{\alpha} = \nu\rangle = \langle F\,\vec{\beta} = \nu'\rangle = \epsilon$.

  Pour le cas (b), supposons $\nu_2 = \emptyset$. Soit ϕ solution de $\langle\nu_1\rangle = \epsilon \bigwedge_{i\in(\!|F|\!)} \alpha_i = \beta_i$. Comme l'invariant est vrai avant application de la règle, ϕ est solution de $\langle F\,\vec{\alpha} = \nu\rangle = \epsilon \bigwedge_{i\in(\!|F|\!)} \alpha_i = \beta_i$ et dans ce contexte, $\phi\,(F\,\vec{\alpha}) =_E \phi\,(F\,\vec{\beta})$. ϕ est donc aussi une solution de $\langle F\,\vec{\alpha} = F\,\vec{\beta} = \nu\rangle = \epsilon \bigwedge_{i\in(\!|F|\!)} \alpha_i = \beta_i$ et on s'est ramené au cas (a) .

— Pour la règle EXPANSE, soit $\nu_0 \subseteq (G\,\vec{\tau} = F\,\vec{\alpha} = \nu)$. On a deux cas à traiter : (a) $\nu_0 \subseteq (F\,\vec{\alpha} = \nu)$ et (b) $G\,\vec{\tau} \in \nu_0$. Dans le cas (a) comme l'invariant est vrai avant la règle, il est vrai après. Si $G\,\vec{\tau} \in \nu_0$, ν_0 est alors de la forme $G\,\vec{\tau} = \nu'_0$. Comme $G\,\vec{\tau}$ est l'expansion de $F\,\vec{\alpha}$, si ϕ est solution de la contrainte $\exists\vec{\beta}.\,\langle G\,\vec{\tau} = \nu'_0\rangle = \langle\nu'\rangle = \epsilon \bigwedge_{i\in(\!|F|\!)} \alpha_i = \beta_i$, alors $\phi\,(F\,\vec{\alpha}) =_E \phi\,(G\,\vec{\tau})$ et ϕ est aussi solution de $\exists\vec{\beta}.\,\langle F\,\vec{\alpha} = \nu'_0\rangle = \langle\nu'\rangle = \epsilon \bigwedge_{i\in(\!|F|\!)} \alpha_i = \beta_i$ et on retombe sur le cas (a).

□

Invariant 5 sur les vues productives

Démonstration. Par cas sur les règles de réécriture qui modifie les vues productives :

- PDispatch : ok car vue singleton.
- PClean : ok car retire un élément de la vue
- Decompose : avant application de la règle, l'invariant est vrai dans les deux vues fusionnées et les deux vues fusionnées partagent un terme avec le même constructeur de tête.
- Expanse : ok car ajoute un terme qui est l'expansion d'un terme dans la vue avant application de la règle.

□

Lemme 1 (1) (préservation des solutions)

- **Merge** : par transitivité de $=_E$ et par l'invariant 2
- **PDispatch** : invariant 4
- **NPDispatch** : soit une solution ϕ de $[\delta] = F\,\vec{\alpha} = \epsilon$, $\phi(F\,\vec{\alpha}) =_E \phi\alpha_i$ et avec l'invariant 2, c'est ok.
- **Cut** : cf [9]
- **Decompose** : il suffit de retirer tous les termes des vues productives.
- **PClean** : par l'invariant 4
- **NPClean** : par l'invariant 2
- **Expanse** : ok car le terme et son expansion sont équivalents.