Déboîter les constructeurs

Nicolas Chataing, Camille Noûs, Gabriel Scherer

Déboîter les constructeurs

Nicolas Chataing[1,2], Camille Noûs[3], and Gabriel Scherer[4]

[1] ENS
[2] Lexifi
[3] Laboratoire Cogitamus
[4] INRIA

Résumé

Nous proposons une implémentation d'une nouvelle fonctionnalité pour OCaml, le déboîtement [1] de constructeurs. Elle permet d'éliminer certains constructeurs de la représentation dynamique des valeurs quand cela ne crée pas de confusion entre différentes valeurs au même type. Nous décrivons :

— un cas d'usage sur les grands entiers où la fonctionnalité améliore les performances de code OCaml idiomatique, éliminant le besoin d'écrire du code non-sûr.
— l'analyse statique nécessaire pour accepter ou rejeter le déboîtement d'un constructeur,
— et l'impact sur la compilation du filtrage par motif.

Pour notre analyse statique, nous devons normaliser certaines expressions de type, avec une relation de normalisation qui ne termine pas nécessairement en présence de types mutuellement récursifs ; nous décrivons une analyse dynamique de terminaison qui garantit la normalisation sans rejeter les définitions de types qui nous intéressent.

1 Introduction

OCaml, ainsi que les autres langages de la famille ML, permet de définir des types qui sont des synonymes, ou abréviations, de types existants, et aussi de nouveaux types de données (sommes ou enregistrements) qui sont distincts de tous les types déjà existants.

```
type an_abbrev = int list
type a_datatype = Short of int | Long of int list
```

Dans l'implémentation de référence du langage, la représentation dynamique (à l'exécution) d'un type somme est la suivante :

— Si elle est constituée d'un constructeur *constant* (sans paramètre), comme `[]` ou `None`, c'est une valeur dite *immédiate* représentée par un entier (`0` pour le premier constructeur constant dans la définition de type, `1` pour le deuxième, etc.).
— Si elle est constituée d'un constructeur *non-constant* (avec un ou plusieurs paramètres), comme `Short n`, elle est représentée par (un pointeur vers) un *bloc* mémoire, débutant avec un *en-tête* suivi par la représentation de chaque paramètre du constructeur. L'en-tête contient une étiquette (tag), le numéro du constructeur (dans l'ordre de définition), et une *arité*, le nombre de paramètres du bloc,

Le filtrage par motif permet de faire une distinction de cas sur les constructeurs d'un type somme, comme dans les deux filtrages suivants :

```
| Some v -> ...
| None -> ...
```

```
| Short n -> ...
| Long li -> ...
```

Le filtrage peut être exécuté efficacement, car un test rapide permet de distinguer des constructeurs différents : deux constructeurs constants sont des entiers différents, deux constructeurs non-constants portent des étiquettes différentes, et OCaml utilise un bit pour distinguer

1. Dans cet article, nous marquons par une couleur particulière les premières apparitions des mots français rigolos exprimant un concept habituellement décrit en anglais, comme le déboîtement (unboxing). Quand le mot n'est pas évident, nous donnons sa traduction d'outre-Manche au premier usage.

les valeurs immédiates (en particulier les constructeurs constants) des blocs (en particulier les constructeurs non-constants).

1.1 Déboîter les types mono-constructeurs

Depuis OCaml 4.06 (juin 2016), OCaml respecte une annotation de déboîtement sur les types sommes contenant un seul cas, dont le constructeur a un seul paramètre (ou sur les enregistrements ayant un seul champ) :

```
type type_name = Type of string [@@unboxed]
type 'a exi = { run: 'b . ('a -> 'b) -> 'b } [@@unboxed]
```

Cette annotation a pour effet que le constructeur (ou l'enregistrement) est effacé de la représentation de la valeur à l'exécution. Par exemple `Type "list"` est représenté exactement comme `"list"`, une valeur de type `string`, au lieu d'être un bloc contenant un paramètre de type `string`. On gagne un pouillème au moment de créer et d'inspecter ces valeurs, et surtout on a une représentation mémoire un peu plus compacte qui peut avoir un impact positif sur les performances sur de gros jeux de données.

En pratique les programmes où cette annotation fait une vraie différence pour les performances sont rares : l'allocation d'un nouveau bloc est très rapide et les programmes bénéficient souvent d'une bonne localité en mémoire, rendant la lecture rapide aussi. L'annotation a surtout l'intérêt de rassurer les programmeurs et programmeuses, en leur donnant la garantie que les nouvelles définitions introduites, pour avoir un type séparé d'un type existant plutôt qu'un synonyme, n'introduisent aucun surcoût.

1.2 Déboîter plus de constructeurs ?

Jeremy Yallop a proposé (Yallop, 2020) d'étendre le déboîtement à certains constructeurs de types sommes ayant plusieurs constructeurs.

```
type a_big_number =
  | Short of int [@unboxed]
  | Long of int list
```

L'annotation `[@@unboxed]` précédente portait sur toute la déclaration, mais ici l'annotation `[@unboxed]` ne porte que sur le constructeur `Short` ; on demande à ce que ce constructeur ne soit pas représenté en mémoire, que `Short n` soit représenté exactement comme l'entier `n`. Cette demande sera acceptée après que le typeur aura vérifié qu'il n'y a pas de confusion possible avec les autres valeurs de ce type, de la forme `Long li`, dont la représentation est toujours distinguable de celle des entiers. Dans le cas général on peut demander à déboîter plusieurs constructeurs du même type, tant que cela n'introduit toujours pas de doublons, deux valeurs différentes ayant la même représentation.

Il faut parfois rejeter cette annotation, car retirer le constructeur introduirait des doublons :

```
type clash_between_int_and_int =
| Int of int [@unboxed]
| Also_int of int [@unboxed]
Error: This declaration is invalid, some [@unboxed] annotations introduce
       overlapping representations.
type t = Constant_constructor_0
type clash_between_constant_constructors =
| T of t [@unboxed]
| Another_constant_constructor_0
Error: This declaration is invalid, some [@unboxed] annotations introduce
```

`overlapping representations.`

Dit autrement : les types *sommes* des langages ML représentent des unions *disjointes*, et le filtrage par motif repose sur le fait de pouvoir distinguer (rapidement) dans quel cas de l'union on se trouve. Une stratégie d'implémentation est d'utiliser des *blocs* et leurs étiquettes pour tous les constructeurs ; cela impose la séparation (disjointness) par construction. Mais dans certains cas, la représentation des valeurs des paramètres porte déjà assez d'information pour les distinguer, et la séparation peut être obtenue sans représenter explicitement les constructeurs.

1.3 Notre approche

— Nous définissons une notion de *tête* d'une valeur qui est une approximation simple de la valeur, efficace à calculer dynamiquement.
— Nous approximons un type par l'ensemble des têtes de leurs valeurs. La représentation des têtes est choisie pour que ces ensembles soient représentables de façon compacte et efficace.
— Nous calculons les ensembles de têtes correspondant à chaque constructeur d'un type de donnée (déboîté ou non), et rejetons la définition si les ensemble des têtes des différents constructeurs ne sont pas disjoints.

Cette approche garantit par construction que les doublons sont rejetés statiquement : si les têtes sont disjointes, les valeurs sont nécessairement disjointes.

Par ailleurs, avec cette approche les têtes suffisent à distinguer les valeurs ; comme elles sont calculables efficacement pendant l'exécution du programme, elles peuvent être utilisées pour exécuter le filtrage par motif.

Dans notre implémentation, la tête d'une valeur est un élément de $\{\texttt{Imm}, \texttt{Block}\} \times \mathbb{Z}_{\texttt{int}}$ (où $\mathbb{Z}_{\texttt{int}}$ est l'ensemble des entiers OCaml de type `int`) défini ainsi :

— La tête d'une valeur immédiate n est la paire $(\texttt{Imm}, n)$
— La tête d'un bloc d'étiquette t est la paire $(\texttt{Block}, t)$.

D'autres choix de têtes sont possibles ; l'approche s'adapte à d'autres langages de programmation ou d'autres implémentations en changeant ce choix. Même au sein d'un langage et d'une représentation dynamique des valeurs, déterminée par une implémentation, on peut choisir une notion de tête plus ou moins fine, qui garde plus ou moins d'information sur la valeur. Une définition moins fine risque de rejeter plus des définitions de types de données, quand les têtes ne sont pas disjointes alors que les valeurs auraient pu l'être. Une définition plus fine permet d'accepter plus de déboîtements de constructeurs, au risque d'être plus coûteuse à calculer pendant le filtrage.

1.3.1 Contribution

Notre contribution, présentée dans cet article, consiste en une implémentation des grandes lignes de la proposition de Jeremy Yallop, dans une version expérimentale du compilateur OCaml [2]. Nous expliquons :

— Une étude de cas où le déboîtement permet un gain de performance, à code égal, ou un gain de propreté et robustesse du code à performances égales.
— Comment décider si une définition doit être rejetée car elle introduit des confusions de représentations.
— En particulier, comment « déplier » des définitions de type pour calculer la représentation des valeurs d'un type donné, en présence de définitions mutuellement récursives qui pour-

2. Nous reprenons les grandes lignes de la spécification de Jeremy Yallop, mais l'avons modifiée pour la rendre plus simple et plus prédictible. Dans cet article nous discutons seulement de notre version, voir Chataing and Scherer (2021) pour une comparaison.

raient nous faire « boucler ».

Les sujets supplémentaires suivants sont traités en annexe :

— Comment adapter le compilateur de filtrage par motif aux de constructeurs déboîtés.
— Des questions sur le support d'autres implémentations utilisant d'autres représentations.
— Un ensemble d'extensions intéressantes qui restent à étudier.

Nous ne traiterons pas dans cet article des questions spécifiques à l'implémentation de ce travail dans le compilateur OCaml.

2 Étude de cas : les grands entiers

Cette nouvelle fonctionnalité est motivée par des considérations de performances. Pour justifier ce travail, voici une étude de cas où elle est utile pour obtenir de bonnes performances.

La bibliothèque Zarith (Miné and Leroy, 2012) fournit une implémentation efficace d'entiers de taille arbitraire, se reposant sur la bibliothèque `Gmp` (Granlund and contributors, 1991) qui fait référence dans ce domaine.

Certains utilisateurs de grands entiers font la majorité de leur calcul sur des entiers plus grands que le type `int` habituel – le type des « petits » entiers. Mais, à l'inverse, une quantité d'utilisateurs font des calculs avec des valeurs que l'on peut presque toujours représenter avec un `int` (sur une architecture avec des mots de 64 bits), mais veulent un résultat précis et mathématiquement exact même dans les cas, rares, de dépassement. Pour cet usage courant, on souhaite minimiser le surcoût par rapport à une implémentation utilisant directement les `int`.

Pour minimiser le surcoût de la précision arbitraire, il est essentiel que le cas courant d'opérations sur des petits entiers ne demande pas d'allocation mémoire, ni d'appel à une fonction C non-triviale (pouvant allouer ou échouer). On veut que la bibliothèque propose un raccourci (fast path) pour les petits entiers.

Zarith implémente des grands entiers avec un type `Zarith.t` qui contient soit des entiers OCaml (comme au type `int`), soit une valeur étrangère (bloc de tag `Custom_tag`). Il n'est pour l'instant pas possible de définir un type ayant cette rerpésentation en OCaml ; l'implémentation de Zarith doit tricher en utilisant les fonctions bas-niveau et non-sûres du module `Obj`, abandonnant la sûreté mémoire ordinairement garantie par typage[3].

```
type t (* int or gmp integer (in a custom block) *)
external is_small_int: t -> bool = "%obj_is_int"
external unsafe_to_int: t -> int = "%identity"
external of_int: int -> t = "%identity"

external c_add: t -> t -> t = "ml_z_add"
let add x y =
  if is_small_int x && is_small_int y then begin
    let z = unsafe_to_int x + unsafe_to_int y in
    (* Overflow check -- Hacker's Delight, section 2.12 *)
    if (z lxor unsafe_to_int x) land (z lxor unsafe_to_int y) >= 0
    then of_int z
    else c_add x y
  end else
    c_add x y
```

3. Une version précédente de Zarith utilisait du code assembleur pour implémenter le raccourci, ce qui permet d'utiliser le drapeau de dépassement présent sur certaines architectures pour détecter efficacement le dépassement. Mais appeler du code assembleur par le mécanisme d'IFE (FFI) OCaml a un coût qui rendait en fait cette version plus lente que la version OCaml actuelle.

Avec notre travail sur les constructeurs déboîtés, il est possible d'écrire le code ainsi :

```
type custom_gmp_t
type t =
  | Short of int [@unboxed]
  | Long of custom_gmp_t
let of_int n = Short n

external c_add: t -> t -> t = "ml_z_add_boxcustom"
let add a b =
  match a, b with
  | Short x, Short y ->
      let z = x + y in
      (* Overflow check -- Hacker's Delight, section 2.12 *)
      if (z lxor x) land (z lxor y) >= 0
      then Short z
      else c_add a b
  | _, _ -> c_add a b
```

Ce code est équivalent à la version précédente quand on prend le raccourci (il génère exactement le même code machine sur cette partie), mais s'écrit en OCaml sûr, sans utiliser `Obj`.

Dans le cas du long chemin (slow path), il reste une différence : cette représentation emboîte les valeurs de type `custom_gmp_t` dans le constructeur `Long`, alors que la version non-sûre les représente directement, sans doublons – ce sont des blocs de tag `Custom_tag`, jamais des entiers.

Notre implémentation des annotations de déboîtement ne permet pas pour l'instant de déboîter le constructeur `Long`, car son argument `custom_gmp_t` est un type abstrait pour le typeur, dont nous supposons qu'il pourrait contenir n'importe quelle valeur. Il est seulement peuplé par le code C, qui y met toujours des valeurs étrangères (tag `Custom`). Nous avons prévu d'étendre à terme notre implémentation pour permettre à l'utilisateur d'annoter ces types abstraits avec des informations (non vérifiables) sur leur structure étrangère, pour permettre de déboîter les deux constructeurs de ce type.

Nous avons fait l'inévitable test de performance sur deux micro-bancs-de-test, d'abord un banc *sans dépassements* qui ne manipule que des petits entiers, et ensuite un banc *avec dépassements* qui construit des entiers `Gmp` pour 16% de ses opérations. (Attention : les deux bancs, avec ou sans dépassements, font des calculs différents, donc la comparaison de leur performance n'a pas de sens.)

version	sans dépassements	16% de dépassements
`int`	6.3s	8.3s (et faux)
`Obj`	1.24x plus lent	2.17x plus lent
`unboxed`	1.24x plus lent	2.38x plus lent
`boxed`	1.56x plus lent	2.41x plus lent

La ligne `int` indique les performances d'une version du code qui n'utilise que des petits entiers, sans aucun test pour détecter les dépassements. Dans le banc de test avec dépassements, elle renvoie un résultat erroné !

La ligne `Obj` indique les performances de la version actuelle de Zarith, écrite avec du code non-sûr. Le surcoût lié aux tests de dépassement est étonnamment faible, 1.24x sans dépassements. Le surcoût de 2.17x dans le banc avec dépassements ne correspond pas aux tests de dépassements, mais au coût des opérations de la bibliothèque `gmp`.

La ligne `boxed` indique les performances d'une version emboitée du code, écrite directement avec un type somme usuel, `type t = Short of int | Long of Gmp.t`.

La ligne `unboxed` indique les performances de notre solution. Sans dépassements, nous sommes dans le régime où notre représentation est identique à celle de Zarith, et les performances sont identiques : l'annotation de déboîtement égale les performances du code non-sûr. En présence de dépassements, notre représentation emboîte les « longs » entiers, et les performances sont proches de la version `boxed`. Une implémentation plus complète que la nôtre, permettant d'annoter les types abstraits avec des informations externes, éliminerait cette dernière différence avec la version `Obj` ; nous pourrions alors recoder Zarith pour utiliser cette fonctionnalité.

Toutes ces mesures ont été effectuées avec le compilateur natif, `ocamlopt`.

Conclusion de l'étude de cas Notre implémentation du déboîtement de constructeurs accélère sensiblement la définition « usuelle » en OCaml, permettant d'égaler les performances du code non-sûr de Zarith pour les programmes manipulant uniquement des petits entiers. Le déboîtement du constructeur de grands entiers est conceptuellement possible, mais n'est pas encore géré par notre prototype.

3 Têtes et formes de tête

Nous devons rejeter les annotations de déboîtement qui introduiraient des doublons, deux valeurs distinctes d'un type se retrouvant avec la même représentation dynamique.

Comme expliqué en introduction (§1.3), nous approximons les valeurs d'un type par leur *tête*, choisie pour être facile à calculer dynamiquement, et pouvoir raisonner statiquement sur les ensembles des têtes correspondant à chaque type.

Dans cette section, nous donnons des têtes une description abstraite, de haut niveau, qui ne dépend pas du langage de programmation – notre traitement s'étend à tous les langages de la famille ML – et des détails d'implémentation.

3.1 Calculer la tête d'un type

Dans cette description de haut niveau, une tête h est soit un constructeur de type somme C of τ, soit un « nom de type primitif » $\widehat{\texttt{t}}$ (par exemple $\widehat{\texttt{int}}$, $\widehat{\texttt{array}}$, $\widehat{\texttt{tuple}}$, $\widehat{\texttt{function}}$), soit $\top$, qui représente n'importe quelle valeur. Une forme de tête (head shape) représente un multi-ensemble de têtes par une liste finie de têtes (notée comme une somme $h_1 + h_2 + \cdots + \emptyset$; on manipule $+$ comme un symbole associatif et on note h pour la somme $h + \emptyset$).

$$h ::= C \text{ of } \tau \mid \widehat{\texttt{t}} \mid \top \qquad\qquad H ::= \emptyset \mid h + H$$

Définissons une petite grammaire d'expressions de types et de définitions de types de données :

$$\begin{array}{lcl} \tau & ::= & \alpha \mid (\tau_i)_i\ t \\ d & ::= & \text{type } (\alpha_i)_i\ t\ =\ (C_j \text{ of } \tau_j)_j\ (C_k^{\text{unboxed}} \text{ of } \tau'_k)_k \end{array}$$

Chaque définition de type somme vient avec une famille (éventuellement vide) de constructeurs emboîtés, et une famille (éventuellement vide) de constructeurs déboîtés.

Notation $((x_i)_{i\in I}, (x_i)_i)$**.** Nous notons $(\tau_i)_{i\in I}$ une famille indicée sur $i \in I$, et souvent $(\tau_i)_i$ quand l'ensemble d'indices n'a pas d'importance.

Nous définissons un jugement $\tau \Rightarrow H$ qui associe une forme de tête H à un type τ. Notre jugement est paramétré par une relation $\widehat{\texttt{T}}$ entre constructeurs de types t et nos têtes primitives $\widehat{\texttt{t}}$. Par exemple on aura typiquement $(\texttt{int}, \widehat{\texttt{int}}) \in \widehat{\texttt{T}}$.

$$\textsc{Var}\ \frac{}{\alpha \Rightarrow \top} \qquad \textsc{Prim}\ \frac{(t, \widehat{\mathtt{t}}) \in \widehat{\mathtt{T}}}{(\tau_i)_i\ t \Rightarrow \widehat{\mathtt{t}}}$$

$$\textsc{Type}\ \frac{\mathsf{type}\ (\alpha_i)_i\ t\ =\ (C_j\ \mathsf{of}\ \_)_j\ (C_k^{\mathsf{unboxed}}\ \mathsf{of}\ \tau'_k)_k \qquad \forall k,\quad \tau'_k[(\alpha_i)_i \leftarrow (\tau_i)_i] \Rightarrow H_k}{(\tau_i)_i\ t \Rightarrow \sum_j C_j + \sum_k H_k}$$

Quand nous calculons la forme de tête d'un type $(\alpha_i)_i\ t$, les variables α_i pourront ensuite être instantiées avec n'importe quel type ; la règle Var leur donne donc la forme $\top$ de toutes les valeurs possibles.

La règle Prim donne leur forme aux types primitifs connus de la relation $\widehat{\mathtt{T}}$.

La règle Type calcule la forme de tête d'un type de donnée, dont les paramètres sont instanciés par des types concrets $(\tau_i)_i$. Elle dépend de la définition de $(\alpha_i)_i\ t$ dans l'environnement de typage, contenant une famille de constructeurs emboîtés C_j et une famille de constructeurs déboîtés C_k^{unboxed}. La forme du type est obtenue en ajoutant aux constructeurs emboîtés C_j l'union des formes des paramètres de type $(\tau'_i)_i$ des constructeurs déboîtés, après avoir remplacés les paramètres α_i par leur instance τ_i.

Considérons par exemple les définitions suivantes :

```
type 'a id = Id of 'a [@unboxed]
type 'a proc =
  | Base of int id [@unboxed]
  | Ask of ('a -> 'a proc) [@unboxed]
  | Tell of 'a * 'a proc
```

Avec une relation $\widehat{\mathtt{T}}$ bien choisie, la forme de tête du type `bool proc` est $\mathtt{Tell} + \widehat{\mathtt{int}} + \widehat{\mathtt{function}}$, comme le justifie la dérivation suivante :

$$\dfrac{\dfrac{\mathsf{type}\ \alpha\ \mathsf{id}\ =\ \mathsf{Id}^{\mathsf{unboxed}}\ \mathsf{of}\ \alpha \qquad \dfrac{\overline{(\mathsf{int}, \widehat{\mathtt{int}}) \in \widehat{\mathtt{T}}}}{\mathsf{int} \Rightarrow \widehat{\mathtt{int}}}}{\mathsf{int\ id} \Rightarrow \widehat{\mathtt{int}}} \qquad \dfrac{\overline{((\mathsf{bool} \to \mathsf{bool\ proc}), \widehat{\mathtt{function}}) \in \widehat{\mathtt{T}}}}{(\mathsf{bool} \to \mathsf{bool\ proc}) \Rightarrow \widehat{\mathtt{function}}}}{\mathsf{bool\ proc} \Rightarrow (\mathtt{Tell}\ \mathsf{of}\ \mathsf{bool} * \mathsf{bool\ proc}) + \widehat{\mathtt{int}} + \widehat{\mathtt{function}}}$$

Hypothèse. Notre jugement dépend d'un choix de type primitifs $\widehat{\mathtt{T}}$ et d'un environnement global de déclarations de types de données. Nous faisons de plus l'hypothèse que les types couverts par ces deux mécanismes sont disjoints : si $(\alpha_i)_i\ t$ est déclaré dans l'environnement, alors aucun $(\tau_i)_i\ t$ n'appartient à $\widehat{\mathtt{T}}$, et réciproquement.

Un type qui n'est ni déclaré ni primitif est traité comme inconnu, il n'y a pas de jugement correspondant. Nous pourrions facilement étendre ce modèle minimal, pour gérer par exemple les abbréviations (avec une règle anologue à Type) et les types abstraits, à qui nous donnons la forme de tête $\top$ – nous parlons des types abstraits plus en détail en Annexe C.2.

3.2 Digression formelle : sémantique des têtes

Pour donner une sémantique précise à ces formes de têtes, il faut inévitablement parler un peu plus des représentations de bas niveau des valeurs.

Hypothèse. Nous supposons donnés un ensemble Data de valeurs de bas niveau, et une *fonction de représentation* $\mathsf{repr} : \mathsf{Value} \to \mathsf{Data}$ ayant la propriété suivante : si $C^{\mathsf{unboxed}}(v)$ est la valeur formée par l'application d'un constructeur déboîté à un argument v, alors on doit avoir

$$\mathsf{repr}(C^{\mathsf{unboxed}}(v)) = \mathsf{repr}(v)$$

Les constructeurs déboîtés sont invisibles dans la représentation des valeurs.

Notation ($\mathcal{M}(S)$, $\{\!\{\ldots\}\!\}$, $\lfloor M \rfloor_{\mathsf{set}}$, $M_1 + M_2$)**.** Nous notons $\mathcal{M}(S)$ l'ensemble des multi-ensembles d'éléments de S, $\{\!\{\ldots\}\!\}$ pour décrire un multi-ensemble plutôt qu'un ensemble, et $\lfloor M \rfloor_{\mathsf{set}}$ l'ensemble des éléments d'un multi-ensemble M : si $M \in \mathcal{M}(S)$, alors $\lfloor M \rfloor_{\mathsf{set}} \in \mathcal{P}(S)$. Nous notons $M_1 + M_2$ pour l'union de deux multi-ensembles.

Définition ($v : \tau$)**.** Nous notons $v : \tau$ quand v est une valeur de type τ. Nous ne spécifions pas le typage des valeurs dans ce document, à part les règles suivantes pour les types de données :

$$\frac{\begin{array}{l}\mathsf{type}\ (\alpha_i)_i\ t\ =\ (C_j\ \mathsf{of}\ \tau_j\ )_j\ (C_k^{\mathsf{unboxed}}\ \mathsf{of}\ \tau'_k)_k \\ v : \tau_j[(\alpha_i \leftarrow \tau''_i)_i]\end{array}}{C_j\ v : (\tau''_i)_i\ t} \qquad \frac{\begin{array}{l}\mathsf{type}\ (\alpha_i)_i\ t\ =\ (C_j\ \mathsf{of}\ \tau_j\ )_j\ (C_k^{\mathsf{unboxed}}\ \mathsf{of}\ \tau'_k)_k \\ v : \tau'_k[(\alpha_i \leftarrow \tau''_i)_i]\end{array}}{C_k^{\mathsf{unboxed}}\ v : (\tau''_i)_i\ t}$$

Définition ($\mathsf{repr}(\tau)$)**.** Nous construisons à partir de notre fonction $\mathsf{repr} : \mathsf{Value} \to \mathsf{Data}$ une fonction $\mathsf{repr} : \mathsf{Type} \to \mathcal{M}(\mathsf{Data})$, qui associe à un type le multi-ensemble des représentations de ses valeurs :

$$\mathsf{repr}(\tau) := \{\!\{\mathsf{repr}(v) \mid v : \tau\}\!\}$$

Hypothèse ($\mathsf{repr}(\widehat{\mathtt{t}})$)**.** On suppose une famille de représentations $\mathsf{repr}(\widehat{\mathtt{t}}) \in \mathcal{P}(\mathsf{Data})$, telle que

$$\forall (v : \tau), \qquad (\tau, \widehat{\mathtt{t}}) \in \widehat{\mathtt{T}} \qquad \Longrightarrow \qquad \mathsf{repr}(v) \in \mathsf{repr}(\widehat{\mathtt{t}})$$

Définition ($\mathsf{repr}(h)$, $\mathsf{repr}(H)$)**.** En plus des représentations primitives $\mathsf{repr}(\widehat{\mathtt{t}})$, nous définissons des représentations $\mathsf{repr}(h), \mathsf{repr}(H) \in \mathcal{M}(\mathsf{Data})$ des têtes et des formes de tête :

$$\begin{array}{lcllcl}\mathsf{repr}(C\ \mathsf{of}\ \tau) & := & \{\!\{\mathsf{repr}(C(v)) \mid v : \tau\}\!\} & \mathsf{repr}(\emptyset) & := & \emptyset \\ \mathsf{repr}(\top) & := & \mathsf{Data} & \mathsf{repr}(h + H) & := & \mathsf{repr}(h) + \mathsf{repr}(H)\end{array}$$

Définition (Sans doublons)**.** On dit d'un type τ (ou d'une forme de tête H) qu'il (elle) est *sans doublons* si le multi-ensemble $\mathsf{repr}(\tau)$ (ou $\mathsf{repr}(H)$) n'a aucun élément répété plusieurs fois.

Deux types ou formes de têtes sont *disjointes* si l'intersection de leur représentation est vide, autrement dit si leur union est sans doublons.

Lemme 1. Si le type τ a la forme de tête H, la représentation de H sur-approxime celle de τ :

$$\tau \Rightarrow H \qquad \Longrightarrow \qquad \mathsf{repr}(\tau) \subseteq \mathsf{repr}(H)$$

En particulier, si H est sans doublons, alors τ est aussi sans doublons.

3.3 Accepter ou rejeter une définition de type

Pour décider si la définition d'un type de donnée $(\alpha_i)_i\ t$ peut introduire des doublons, on calcule la forme de tête de $(\alpha_i)_i\ t$:

$$\textsc{Typedecl}\ \frac{(\alpha_i)_i\ t \Rightarrow R}{(\mathsf{type}\ (\alpha_i)_i\ t\ =\ \ldots) \Rightarrow R}$$

et on regarde si cette forme contient des conflits : on rejette la déclaration si la forme de tête H contient des têtes dont les représentations de bas niveau ne sont pas disjointes, c'est-à-dire si $\mathsf{repr}(H)$ est un multi-ensemble contenant des doublons.

Remarquons que, dans ce test, l'expression de type $(\alpha_i)_i\ t$ contient des variables libres α_i. Si ces variables interviennent pendant le calcul de la forme de tête, on obtient $\top$ dans la forme de tête calculée. Ce n'était pas le cas dans notre exemple bool proc, qui ne contient pas de constructeur $C^{\mathsf{unboxed}}\ \mathsf{of}\ \alpha_i$.

À l'inverse, quand calcule la forme de tête d'un type, on rencontre des expressions de type comme int id dans notre exemple où les paramètres ont une instance concrète. On calcule donc une forme de tête pour ce type ($\widehat{\texttt{int}}$) qui est plus fine que la forme de tête $\top$ du cas général $\alpha\,\mathsf{id}$. Prendre ainsi en compte les paramètres de type permet d'accepter plus de définitions : si nous approximions la forme de tête de int id par celle de α id, c'est-à-dire $\top$, la définition de α proc ci-dessus serait rejetée par notre critère, puisque sa forme de tête $(\texttt{Tell}\ \mathsf{of}\ \ldots) + \top + \widehat{\texttt{function}}$ contient des doublons : l'ensemble de valeurs correspondant à $\top$ a une intersection non vide avec celui de `Tell` of ... et celui de $\widehat{\texttt{function}}$.

Cette approche n'est pas modulaire, elle repose sur la disponibilité et le dépliage systématique des définitions de type. Cela ne pose pas de problème de performances car le nombre de définitions de types reste petit en pratique ; nous discutons modularité en Annexe C.3.

4 Contrôle dynamique de la terminaison

4.1 Intuition

Le jugement $\tau \Rightarrow H$ donne une spécification du calcul de la forme de tête, mais pas un algorithme effectif pour la calculer, à cause des problèmes de terminaison en présence de types de données mutuellement récursifs. Considérons par exemple la définition

```
type 'a t = Foo | Loop of 'a t [@unboxed]
```

Calculer naïvement la forme de tête de `int t` (par exemple) conduirait à une boucle infinie, puisque la règle Type suggère de calculer la forme de tête du constructeur déboîté `Loop`, à savoir `int t` de nouveau.

Dit autrement, notre algorithme revient à calculer la forme normale d'expression de types pour une relation d'expansion / de récriture dépliant certaines définitions de type. Il n'existe pas toujours de forme normale, ce processus peut ne pas terminer dans le cas de définitions de types mutuellement récursives.

Interdire statiquement les cycles, comme pour les abréviations ? Dans le cas des définitions d'abréviations/synonymes (les définitions de la forme $\mathsf{type}\ (\alpha_i)_i\ t = \tau$, qui donnent un nouveau nom à un type existant au lieu de définir un nouveau type de donnée), cette situation n'apparaît pas car OCaml interdit déjà les abréviations cycliques. Dans un groupe de types mutuellement récursifs, contenant des abréviations et des types de donnée, tout cycle dans la relation « la définition de `t` mentionne le constructeur de type `u` » doit être « cassé » par au moins une définition de type de données, sinon le groupe de définitions est rejeté. Mais nous ne pouvons pas traiter les constructeurs déboîtés comme les abréviations, car cela voudrait dire qu'ajouter une annotation `[@unboxed]` pourrait transformer un cycle autorisé en un cycle interdit dans de nombreux cas utiles. Par exemple :

```
type 'a thunk = unit -> 'a
type 'a stream =
| Next of ('a * 'a stream) thunk [@unboxed]
| End
```

Cette définition décrit un type de flots (stream) paresseux, qui utilise une abréviation auxiliaire `'a thunk`. Le type `'a stream` est récursif, il apparaît dans le type du constructeur `Next`. Sans annotation de déboîtement, ces définitions sont parfaitement valides. Une approche qui rejette

tous les cycles sauf ceux qui passent par au moins constructeur emboîté n'est pas satisfaisante.

Détecter la répétition d'un type pendant l'expansion ? Une idée naturelle est de garder trace, pendant le calcul de la forme de tête d'un type, des expressions de type dont on est en train de calculer la forme de tête. Si une règle demande de calculer de nouveau l'un des types déjà présents dans cette trace (dans notre exemple : calculer la forme de `int t` pour calculer celle de `int t`), on a trouvé un cycle et le calcul peut s'arrêter avec une erreur.

Remarque. On parle ici d'un test *dynamique* de terminaison, même si le test est effectué pour calculer la forme de tête d'un type, donc pendant le typage du programme fourni par l'utilisateur. En effet, il ne s'agit pas d'une analyse statique qui décide a priori d'accepter ou rejeter des définitions mutuellement récursive, on détecte la non-terminaison *pendant* qu'on déplie les définitions pour en calculer la forme de tête. C'est l'équivalent d'une « instrumentation » ou surveillance (monitoring) d'un programme qui collecte des informations pendant son exécution, pour l'interrompre s'il risque de ne pas respecter une certaine propriété de sûreté.

Hélas, cette méthode ne garantit pas la terminaison en présence de paramètres de type utilisés de façon dite *non-régulière*, comme dans cet exemple :

```
type 'a t = Loop of ('a list) t [@unboxed]
```

Ici, demander la forme de tête de `int t` conduit à demander celle de `int list t`, puis de `int list list t`, etc. Il y a une boucle infinie, mais avec des expressions de type de plus en plus grosses, chacune distincte de toutes les précédentes.

Détecter la répétition d'un constructeur de type pendant l'expansion ? Dans l'exemple précédent, aucune expression de type ne se répète, mais le constructeur de type `t` se répète en tête de chaque type de la trace, et c'est son expansion qui provoque la non-terminaison. On pourrait rendre le test précédent moins fin (rejeter plus de programmes) en arrêtant avec une erreur les expansions qui répètent le même constructeur de tête plusieurs fois.

Cependant, ce critère est maintenant trop grossier, il rejette des définitions sûres et intéressantes. Voici deux exemples, l'un synthétique et l'autre plus réaliste.

```
type 'a id = Id of 'a [@unboxed]
type t = Foo of int id id [@unboxed]

type 'a located = 'a * Location.t
type expr = expr_ located
and expr_ = ... | Name of name [@unboxed] | ...
and name = string located
```

Dans le premier exemple, calculer la forme de tête de `int id id` demande deux expansions successives du constructeur de tête `id`, et serait rejeté par le critère que nous proposons. Le second exemple est plus réaliste ; il montre que des types paramétrés comme `'a id`, ici `'a located`, peuvent être utilisés pour factoriser des aspects spécifiques d'une structure de données complexe, être utilisés dans plusieurs définitions mutuellement récursives (ici les « expressions » et les « noms » d'un petit langage), et parfois se retrouver plusieurs fois le long d'un chemin de définitions à déplier.

Notre solution : garder une trace par sous-expression Notre approche consiste à détecter la répétition d'un constructeur dans une trace, mais en gardant des traces « locales » : chaque sous-expression de type garde trace des expansions qui ont eu lieu avant son apparition, mais pas des expansions qui ont suivi. Cela suffit pour garantir la terminaison (comme nous le verrons ensuite), mais cela autorise les exemples précédents utilisant `int id id` ou `'a located`.

4.2 Digression formelle : la forme de tête comme une forme normale

Les définitions de types (types sommes et abréviations) des langages ML peuvent être vus comme des termes d'un lambda-calcul simplement typé comportant des définitions globales, mutuellement récursives. Par exemple, les définitions suivantes

```
type 'a id = Id of 'a [@unboxed]
type 'a proc =
  | Base of int id [@unboxed]
  | Ask of 'a -> 'a proc [@unboxed]
  | Tell of 'a * 'a proc
```

Peuvent être vues comme les déclarations suivantes :

$$\mathsf{id} := \lambda\alpha.\ \mathsf{Id}^{\mathsf{unboxed}}\ \mathsf{of}\ \alpha$$

$$\mathsf{proc} := \lambda\alpha.\ (\mathsf{Base}^{\mathsf{unboxed}}\ \mathsf{of}\ (\mathsf{int}\ \mathsf{id})) + (\mathsf{Ask}^{\mathsf{unboxed}}\ \mathsf{of}\ (\alpha, \alpha\ \mathsf{proc})\ (\to)) + (\mathsf{Tell}\ \mathsf{of}\ (\alpha, \alpha\ \mathsf{proc})\ (\times))$$

avec des fonctions n-aires dont l'application se note à l'envers $(\tau_i)_i\ t$ (la fonction t appliquée à une famille d'arguments $(\tau_i)_i$), et où les constructeurs de termes $\mathsf{Base}^{\mathsf{unboxed}}\ \mathsf{of}\ \tau$, certains noms de fonctions primitifs $(\to)$, $(\times)$, et la forme binaire $t + u$ sont vus comme des neutres dont l'application ne se réduit pas.

$$\tau := \alpha \mid C\ \mathsf{of}\ \tau \mid C^{\mathsf{unboxed}}\ \mathsf{of}\ \tau \mid (\tau_i)_i\ t \mid \tau + \tau'$$

En particulier, l'expansion d'un constructeur de tête, remplacé par sa définition où l'on remplace les instances de ses paramètres de type, correspond à la règle de β-réduction usuelle :

$$\frac{t := \lambda(\alpha_i)_i.\ \tau'}{(\tau_i)_i\ t \rightsquigarrow_\beta \tau'[(\alpha_i \leftarrow \tau_i)_i]}$$

Avec cette présentation, la forme de tête d'une expression de type peut se lire depuis la *forme normale* de cette expression pour une certaine stratégie de réduction :

— on réduit sous les sommes $t + u$
— on ne réduit pas sous les applications des constructeurs types de base : $(\times)$, $(\to)$, etc.
— on réduit sous les constructeurs déboîtés C^{unboxed}
— on ne réduit pas sous les constructeurs emboîtés

On définit les contextes de réduction $E[\Box]$ ci-dessous, et une notion de forme normale V correspondante :

$$E ::= \Box \mid C^{\mathsf{unboxed}}\ \mathsf{of}\ \tau \mid \tau + E \mid E + \tau$$
$$V ::= \alpha \mid C\ \mathsf{of}\ \tau \mid C^{\mathsf{unboxed}}\ \mathsf{of}\ V \mid V + V'$$

$$\frac{\tau \rightsquigarrow_\beta \tau'}{E[\tau] \rightsquigarrow E[\tau']}$$

La forme normale résultante V n'est pas exactement une forme de tête H, elle porte plus d'information. Pour récupérer exactement la forme de tête, on effectue une simple opération d'effacement $\lfloor V \rfloor$:

$$\begin{array}{lcl} \lfloor \alpha \rfloor & := & \top \\ \lfloor C\ \mathsf{of}\ \tau \rfloor & := & C\ \mathsf{of}\ \tau \\ \lfloor C^{\mathsf{unboxed}}\ \mathsf{of}\ V \rfloor & := & \lfloor V \rfloor \\ \lfloor V + V' \rfloor & := & \lfloor V \rfloor + \lfloor V' \rfloor \end{array} \qquad \frac{((\tau_i)_i\ t,\ \widehat{\mathtt{t}}) \in \widehat{\mathtt{T}}}{\lfloor (\tau_i)_i\ t \rfloor := \widehat{\mathtt{t}}}$$

Par exemple, la forme normale de $\mathsf{bool}\,\mathsf{proc}$ vu comme un λ-terme, pour cette stratégie, est : $V := \mathsf{Base}^{\mathsf{unboxed}}\mathsf{of}(\mathsf{Id}^{\mathsf{unboxed}}\ \mathsf{of}\ \mathsf{int}) + \mathsf{Ask}^{\mathsf{unboxed}}\mathsf{of}(\mathsf{bool}, \mathsf{bool}\ \mathsf{proc})\ (\to) + \mathsf{Tell}\mathsf{of}(\mathsf{bool}, \mathsf{bool}\ \mathsf{proc})\ (\times)$

et l'effacement décrit ci-dessus nous redonne exactement sa forme de tête :

$$\lfloor V \rfloor = \widehat{\texttt{int}} + \widehat{\texttt{function}} + \mathsf{Tell\ of\ bool} \times \mathsf{bool\ proc}$$

Lemme 2. On a $\tau \Rightarrow H$ si et seulement τ a une forme normale V qui s'efface en H.

$$\tau \Rightarrow H \qquad \Longleftrightarrow \qquad \exists V, \quad \tau \rightsquigarrow^* V \quad \wedge \quad \lfloor V \rfloor = H$$

Remarque. Avec la même β-réduction, mais d'autres stratégies de réduction et d'autres analyses des formes normales résultantes, on peut représenter d'autres analyses des expressions de type qui ont besoin de déplier les définitions. Y compris des analyses déjà existantes en OCaml, par exemple : « est-ce que ce type ne contient que des valeurs immédiates ? ». Notre Théorème 1 de terminaison s'applique aussi dans le cas d'une relation de réduction maximale où on peut réduire en profondeur les arguments d'un constructeur, ou sous les constructeurs emboîtés, donc il s'applique en fait à toutes les analyses possibles sur des formes normales de cette grammaire.

4.3 Types annotés par des traces

Nous définissons une nouvelle grammaire d'expressions de types « annotées » $\bar{\tau}$, où chaque expression et sous-expression de type est une paire $\tau \mathbin{@} l$ contenant une trace d'expansion l, qui est une liste de constructeurs de type t.

$$\bar{\tau} \quad ::= \quad \tau \mathbin{@} l \qquad\qquad \tau \quad ::= \quad \alpha \mid (\bar{\tau}_i)_i\, t \qquad\qquad l \quad ::= \quad \emptyset \mid l, t$$

Un exemple de type annoté est $(\alpha \mathbin{@} l_\alpha, \beta \mathbin{@} l_\beta)\, t \ @\ l_t$. L'expression en tête applique le constructeur de type t, avec la trace l_t, et les deux paramètres de t sont aussi annotés avec les traces l_α et l_β.

Remarque. Nous réutilisons le non-terminal τ dans cette grammaire par analogie avec la grammaire τ des expressions de type usuelles, mais ici τ désigne une expression de type dont les sous-expressions sont annotées. Ce que nous voulons dire par τ devrait être clair selon le contexte ; dans la suite du document nous n'utilisons pas la méta-variable $\bar{\tau}$ mais toujours explicitement la paire $\tau \mathbin{@} l$, et les usages « annotés » de τ sont toujours de cette forme.

On peut maintenant raffiner la relation de réduction sur les types annotés, pour formaliser notre algorithme de contrôle dynamique de la terminaison : avant de déplier une définition de type de donnée, on vérifie que le constructeur correspondant n'apparaît pas déjà dans la trace de l'application du constructeur, et sinon on réduit vers un nouveau terme d'erreur `Cycle`.

$$\bar{\tau}_\perp \quad ::= \quad \bar{\tau} \mid \texttt{Cycle} \qquad \frac{t := \lambda(\alpha_i)_i.\ \tau' \qquad t \notin l}{(\tau_i)_i\, t \mathbin{@} l \rightsquigarrow_\beta \tau'[(\alpha_i \leftarrow \tau_i)_i]^{@l,t}} \qquad \frac{t := \lambda(\alpha_i)_i.\ \tau' \qquad t \in l}{(\tau_i)_i\, t \mathbin{@} l \rightsquigarrow_\beta \texttt{Cycle}}$$

$$V_\perp \quad ::= \quad V \mid \texttt{Cycle} \qquad \frac{\tau \rightsquigarrow_\beta \tau'}{E[\tau] \rightsquigarrow E[\tau']} \qquad \frac{\tau \rightsquigarrow_\beta \texttt{Cycle}}{E[\tau] \rightsquigarrow \texttt{Cycle}}$$

La nouvelle définition de la β-réduction remplace la substitution d'expressions de types non-annotées $\tau'_k[(\alpha_i)_i \leftarrow (\tau_i)_i]$ par une « substitution annotante » $\tau'_k[(\alpha_i)_i \leftarrow (\tau_i \mathbin{@} l_i)_i]^{@l,t}$, une méta-opération $\tau[\sigma]^{@l}$ définie ainsi :

$$\begin{aligned} \alpha[\sigma]^{@l} &= \sigma(\alpha) \\ ((\tau_i)_i\, t)[\sigma]^{@l} &= \Big((\tau_i[\sigma]^{@l})\, t\Big) \mathbin{@} l \end{aligned}$$

La substitution annotante $\tau[\sigma]^{@l}$ prend une expression de type non-annotée τ, une substitution σ de variables de type libres vers des expressions de types annotées $\tau_i \mathbin{@} l_i$, et une « trace courante » l. Elle renvoie une expression de type annotée qui utilise les $\tau_i \mathbin{@} l_i$ à la place des variables α, et qui sur toutes les autres sous-expressions de type utilise la trace l. Par exemple :

$$(\alpha \to \mathsf{int})[\alpha \leftarrow \tau_\alpha \mathbin{@} l_\alpha]^{@l} \qquad = \qquad (\tau_\alpha \mathbin{@} l_\alpha \to \mathsf{int} \mathbin{@} l) \mathbin{@} l$$

L'effet de cette substitution annotante dans la règle de β-réduction est le suivant. Quand on expanse une version annotée de $(\tau_i)_i\, t$, les arguments τ_i du constructeur de type t sont déjà annotés, et ils conservent leur annotation dans l'expansion. Par contre, les « nouvelles » sous-expressions de type, qui viennent de la définition du type de donnée t qui s'expanse en un type paramétré τ', ne portaient pas d'annotation. On leur donne, dans le résultat, la trace courante de l'expansion, l, à laquelle on ajoute le constructeur t que l'on vient d'expanser.

Ainsi, à tout moment du calcul, chaque sous-expression de type est annotée par la trace des expansions effectuées jusqu'à son apparition, par expansion d'un constructeur de type.

On peut enfin définir un jugement $\tau \Rightarrow_\bot H_\bot$ de mise en forme de tête contenant ce contrôle de la terminaison. Ci-dessous, l'expression $\tau[\emptyset]^{@\emptyset}$ correspond à annoter en profondeur un type non-annoté τ avec la trace vide $\emptyset$.

$$H_\bot ::= H \mid \texttt{Cycle} \qquad\qquad \lfloor \texttt{Cycle} \rfloor := \texttt{Cycle} \qquad\qquad \frac{\tau[\emptyset]^{@\emptyset} \rightsquigarrow^\star V_\bot}{\tau \Rightarrow_\bot \lfloor V_\bot \rfloor}$$

Cette stratégie de contrôle de la terminaison est *correcte* : les calculs qu'elle produit sans lever l'erreur `Cycle` correspondent à des réductions dans le système non-annoté de départ.

Lemme 3 (Correction du contrôle de terminaison)**.**
Si $\tau \mathbin{@} l \rightsquigarrow^\star \tau' \mathbin{@} l'$, alors les termes non-annotés τ, τ' correspondant sont aussi dans la relation $\tau \rightsquigarrow^\star \tau'$. Par conséquent, si $\tau \Rightarrow_\bot H$ alors $\tau \Rightarrow H$.

Remarque. Notre implémentation n'utilise pas une relation petit-pas comme ici, mais une fonction récursive plus proche d'une relation grand-pas comme le jugement initial : on utilise partout des expressions de type annotées $\tau \mathbin{@} l$, et on utilise la même substitution annotante dans la règle Type. La présentation petit-pas, équivalente, est choisie ici car elle facilite la présentation de la preuve de terminaison dans la section suivante.

Exemple avec cycle Dans le cas de la définition

```
type 'a loop = Loop of 'a list loop [@unboxed]
```

calculer la forme de tête de ce type de donnée demande de normaliser le type annoté

$$(\alpha \mathbin{@} \emptyset)\ \mathsf{loop} \mathbin{@} \emptyset$$

Une étape de β-réduction donne l'expression de type annotée suivante :

$$(\alpha' \ \mathsf{list})\ \mathsf{loop}[\alpha' \leftarrow \alpha \mathbin{@} \emptyset]^{@[\mathsf{loop}]} \qquad = \qquad ((\alpha \mathbin{@} \emptyset)\ \mathsf{list} \mathbin{@} [\mathsf{loop}])\ \mathsf{loop} \mathbin{@} [\mathsf{loop}]$$

(Nous notons $[\mathsf{loop}]$ pour la trace $(\emptyset, \mathsf{loop})$.) Dans cette expression, α garde son annotation de départ, mais les deux nouveaux constructeurs de type list et loop sont maintenant annotés avec une trace qui indique qu'ils ont été introduits par l'expansion du constructeur loop.

Il n'est plus possible d'expanser ce terme car la prémisse $t \notin l$ est invalide : le constructeur loop apparaît dans la trace $[\mathsf{loop}]$. La seule réduction possible est vers le terme d'erreur `Cycle`.

$$\frac{(\alpha\ \mathsf{loop})[\emptyset]^{@\emptyset} = (\alpha \mathbin{@} \emptyset)\ \mathsf{loop} \mathbin{@} \emptyset \rightsquigarrow ((\alpha \mathbin{@} \emptyset)\ \mathsf{list} \mathbin{@} [\mathsf{loop}])\ \mathsf{loop} \mathbin{@} [\mathsf{loop}] \rightsquigarrow \texttt{Cycle}}{\left(\mathsf{type}\ \alpha\ \mathsf{loop}\ =\ \mathsf{Loop}^{\mathsf{unboxed}}\ \mathsf{of}\ \alpha\ \mathsf{list}\ \mathsf{loop}\right) \Rightarrow \texttt{Cycle}}$$

Exemple sans cycle Reprenons notre exemple de l'expression de type problématique `int id id` :

```
type 'a id = Id of 'a [@unboxed]
type t = Foo of int id id [@unboxed]
```

Calculer la forme de tête de `t` revient à normaliser l'expression annotée

$$((\mathsf{int} \mathbin{@} \emptyset)\ \mathsf{id} \mathbin{@} \emptyset)\ \mathsf{id} \mathbin{@} \emptyset$$

qui se réduit en

$$\alpha'[\alpha' \leftarrow (\mathsf{int} \mathbin{@} \emptyset)\ \mathsf{id} \mathbin{@} \emptyset]^{@[\mathsf{id}]} \qquad = \qquad (\mathsf{int} \mathbin{@} \emptyset)\ \mathsf{id} \mathbin{@} \emptyset$$

L'expression de type que nous obtenons vient de l'expansion de id, mais les constructeurs de type qui la composent ne sont pas « nouveaux », ils viennent du paramètre de l'expression de départ où ils étaient annotés, et gardent la même annotation. En particulier, l'expression de type résultante ne porte pas la trace [id], qui empêcherait d'y expanser de nouveau id, mais la trace initiale $\emptyset$. Nous pouvons faire une nouvelle expansion vers une forme normale : int @ $\emptyset$.

$$\frac{((\mathsf{int}\ \mathsf{id})\ \mathsf{id})[\emptyset]^{@\emptyset} = ((\mathsf{int} \mathbin{@} \emptyset)\ \mathsf{id} \mathbin{@} \emptyset)\ \mathsf{id} \mathbin{@} \emptyset \rightsquigarrow (\mathsf{int} \mathbin{@} \emptyset)\ \mathsf{id} \mathbin{@} \emptyset \rightsquigarrow \mathsf{int} \mathbin{@} \emptyset \qquad \dfrac{(\mathsf{int}, \widehat{\texttt{int}}) \in \widehat{\mathsf{T}}}{\lfloor \mathsf{int} \rfloor = \widehat{\texttt{int}}}}{\left(\mathsf{type\ t} \ = \ \mathsf{Foo}^{\mathsf{unboxed}}\ \mathsf{of\ int\ id\ id}\right) \Rightarrow \widehat{\texttt{int}}}$$

4.4 Preuve de terminaison

Nous présentons notre argument de terminaison dans un cadre plus abstrait de récriture d'arbres : des arbres n-aires dont les nœuds sont étiquetés, avec un process d'expansion d'un arbre vers un nouvel arbre, et une relation d'ordre bien fondée sur les noeuds qui garantit la terminaison des récritures par expansion.

4.4.1 Arbres n-aires

Définition (Arbres n-aires ouverts)**.** Étant donné deux ensembles $\mathcal{N}$ (les étiquettes de noeuds) et $\mathcal{V}$ (les variables), on définit les arbres n-aires ouverts $a \in \mathsf{Tree}(\mathcal{N}, \mathcal{V})$ par la grammaire :

$$\begin{array}{rcll} a \in \mathsf{Tree}(\mathcal{N}, \mathcal{V}) & ::= & \mathsf{node}(c,\ (a_i)_i) & (c \in \mathcal{N}, i \in \{1, 2, \ldots, n\}) \\ & | & \mathsf{var}(x) & (x \in \mathcal{V}) \end{array}$$

Un nœud d'un arbre est formé soit d'un constructeur $c \in \mathcal{N}$ suivi d'un nombre arbitraire (fini) de sous-arbres, soit par une variable $x \in \mathcal{V}$.

Définition (join)**.** Ces arbres ont une structure monadique, on peut définir une opération

$$\mathsf{join} : \mathsf{Tree}(\mathcal{N}, \mathcal{V} \uplus \mathsf{Tree}(\mathcal{N}, \mathcal{V})) \to \mathsf{Tree}(\mathcal{N}, \mathcal{V})$$

$$\frac{}{\mathsf{join}(\mathsf{node}(c,\ (a_i)_i)) := \mathsf{node}(c,\ (\mathsf{join}(a_i))_i)} \qquad \frac{x \in \mathcal{V}}{\mathsf{join}(\mathsf{var}(x)) := \mathsf{var}(x)} \qquad \frac{x \in \mathsf{Tree}(\mathcal{N}, \mathcal{V})}{\mathsf{join}(\mathsf{var}(x)) := x}$$

Définition (π, $b \in_\pi a$, $\mathsf{Subtree}(a)$, $\pi.\pi'$, $\pi' \geq \pi$)**.** On définit un chemin π, la relation $b \in_\pi a$ si π est un chemin du sous-arbre b à son parent a, et le multi-ensemble $\mathsf{Subtree}(a)$ des sous-arbres de a.

$$\begin{array}{rcll} \pi & ::= & \emptyset & \\ & | & \pi.i & (i \in \mathbb{N}) \end{array} \qquad \frac{}{a \in_\emptyset a} \qquad \frac{b \in_\pi a_i}{b \in_{\pi.i} \mathsf{node}(c,\ (a_i)_i)}$$

$$\mathsf{Subtree}(a) := \{\!\{ b \mid \exists \pi,\ b \in_\pi a \}\!\}$$

On peut remarquer que la relation $b \in_\pi a$ est compatible avec la définition naturelle de la concaténation de chemins : si $a_1 \in_{\pi_{12}} a_2$ et $a_2 \in_{\pi_{23}} a_3$, alors $a_1 \in_{\pi_{12}.\pi_{23}} a_3$.

Enfin, on dit que π' est une extension de π, notée $\pi' \geq \pi$, si π' est de la forme $\pi''.\pi$.

$$\pi' \geq \pi \qquad := \qquad \exists \pi'',\ \pi' = \pi''.\pi$$

Définition ($a[\pi \leftarrow b]$)**.** $a[\pi \leftarrow b]$ est l'arbre a où le sous-arbre en π est remplacé par b.

$$\frac{}{a[\emptyset \leftarrow b] := b} \qquad \frac{a_j[\pi \leftarrow b] = a'}{\mathsf{node}(c,\, (a_i)_{i\in I})[\pi.j \leftarrow b] := \mathsf{node}(c,\, \Big((a_i)_{i\in J\setminus\{i\}}, (j \mapsto a')\Big))}$$

Ce cadre général nous intéresse car il modélise notre calcul de forme de tête.

Exemple (Arbres de types annotés)**.** Nous considérons comme des arbres n-aires les expressions de type annotées $\tau \mathbin{@} l$: ces expressions se plongent dans l'ensemble $\mathsf{Tree}(\mathcal{T}, \emptyset)$ pour un ensemble de constructeurs de noeuds $\mathcal{T}$ ayant la grammaire suivante :

$$c ::= \alpha \mathbin{@} l \mid t \mathbin{@} l$$

Définition ($\mathsf{tree}(\tau)$)**.** Nous notons $\mathsf{tree}(\tau)$ pour l'arbre correspondant au type annoté τ.

$$\mathsf{tree}(\alpha \mathbin{@} l) := \mathsf{node}(\alpha \mathbin{@} l,\, \emptyset) \qquad \mathsf{tree}((\bar\tau'_i)_i \mathbin{@} l) := \mathsf{node}(t \mathbin{@} l,\, (\mathsf{tree}(\bar\tau'_i))_i)$$

Par exemple l'expression de type annotée $(\alpha \mathbin{@} l_\alpha, \beta \mathbin{@} l_\beta)\, t \mathbin{@} l_t$ devient l'arbre
$\mathsf{node}(t \mathbin{@} l_t,\, (0 \mapsto \mathsf{node}(\alpha \mathbin{@} l_\alpha, \emptyset), 1 \mapsto \mathsf{node}(\beta \mathbin{@} l_\beta, \emptyset)))$

4.4.2 Expansion d'arbre

Nous allons maintenant définir une notion d'expansion sur les arbres, qui généralise l'expansion des définitions dans les expressions de type. Informellement, nous considérons le processus d'expansion suivant :

- Un nœud (en position arbitraire) de l'arbre est choisi pour être expansé ; le sous-arbre dont il est racine est appelé le « sous-arbre expansé ».
- Le processus d'expansion remplace le sous-arbre expansé par un sous-arbre contenant :
  - zéro, un ou plusieurs « nouveaux » nœuds qui ne correspondent pas à une occurrence d'un noeud dans l'arbre de départ, et
  - un nombre arbitraire de copies de certains sous-arbres stricts du nœud expansé.

Un sous-arbre du nœud expansé peut disparaître pendant l'expansion, ou être dupliqué en plusieurs sous-arbres, placés à l'intérieur du sous-arbre remplaçant le sous-arbre expansé.

Définition (Expansion d'arbre)**.** Une expansion d'arbre *en tête* $b \rightsquigarrow_\beta b'$ consiste à remplacer un arbre b par un arbre b' dont certains sous-arbres sont des sous-noeuds *stricts* de b.

$$\frac{b \in \mathsf{Tree}(\mathcal{N}, \mathcal{V}) \qquad b_s \in \mathsf{Tree}(\mathcal{N}, \mathcal{V} \uplus (\mathsf{Subtree}(b) \setminus \{b\}))}{b \rightsquigarrow_\beta \mathsf{join}(b_s)}$$

Une expansion d'arbre $a \rightsquigarrow a'$ est une expansion en tête d'un sous-arbre de a.

$$\frac{b \in_{\pi_b} a \qquad b \rightsquigarrow_\beta b'}{a \rightsquigarrow a[\pi_b \leftarrow b']}$$

Exemple. La Figure 4.4.2 représente un arbre à gauche, et trois expansions possibles de cet arbre à droite, toutes à partir du nœud c, où les « nouveaux nœuds » sont montrés en rouge. Dans la première expansion, le nœud c est remplacé par un nouveau nœud c', auquel sont attachés les enfants du nœud c (ni effacés ni dupliqués). Dans la seconde expansion, il n'y a pas de nouveau nœud, il ne reste à la place de c que son sous-arbre de racine e (et d est effacé). Dans la troisième expansion, il y a deux nouveau nœuds c' et c'', deux copies du sous-arbre de racine e, une copie du sous-arbre de racine f, et d est effacé.

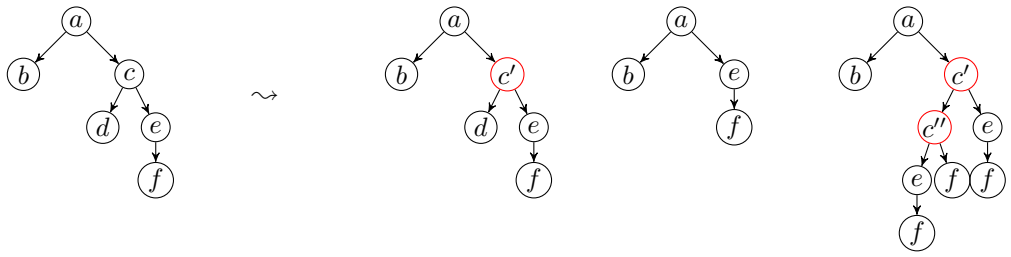

FIGURE 1 – Expansions de sous-arbres

Exemple. Le processus d'expansion d'arbre généralise l'expansion des définitions dans une expression de type : si $\tau \rightsquigarrow \tau'$, alors $\mathsf{tree}(\tau) \rightsquigarrow \mathsf{tree}(\tau')$.

En particulier, à toute séquence infinie d'expansions d'une expression de type correspond une séquence infinie d'expansions sur l'arbre correspondant.

Remarque. Avec nos expressions de type annotées, un type paramétré $(\tau_i)_i\ t$ s'expanse en remplaçant des variables de la définition de t par les paramètres τ_i. Au niveau des arbres cela correspond à des expansions où les « vieux » sous-arbres sont toujours les enfants directs du nœud expansé, et pas des sous-arbres stricts arbitraires. Mais cette notion plus riche d'expansion est utile pour modéliser les contraintes de type du langage OCaml

```
type 'a t = ('b * bool) constraint 'a = 'b * 'b * int
```

qui permettent effectivement de désigner des sous-expressions de type arbitraires.

Définition ($\mathsf{Tree}(\mathcal{N})$)**.** L'ensemble des variables $\mathcal{V}$ est utile pour définir l'expansion par substitution. On ne s'en servira plus par la suite : on notera $\mathsf{Tree}(\mathcal{N})$ pour $\mathsf{Tree}(\mathcal{N}, \emptyset)$.

Définition ($\mathsf{constr}(a)$)**.** Pour $a \in \mathsf{Tree}(\mathcal{N})$, on définit $\mathsf{constr}(a)$ le constructeur de tête de a (qui ne peut pas être une variable, puisqu'on est sur $\mathsf{Tree}(\mathcal{N}, \emptyset)$:

$$\mathsf{constr}(\mathsf{node}(c,\ (a_i)_i)) := c$$

4.4.3 Terminaison de l'expansion

Définition (Mesure)**.** Si A est un ensemble, on appelle *mesure sur* A le choix d'une fonction $m_A : A \to M$ pour un ensemble $(M, <_M)$ muni d'un ordre bien fondé [4].

Définition (Expansion mesurée)**.** Étant donné une mesure $m_{\mathcal{N}}$ sur $\mathcal{N}$, on dit qu'une expansion $a \rightsquigarrow a' \in \mathsf{Tree}(\mathcal{N})$ est *mesurée* si les « nouveaux » nœuds de l'expansion ont une mesure

4. Un ordre $(<)$ est bien fondé s'il n'existe pas de chaîne infinie descendante, de suite $(m_i)_{i\in\mathbb{N}}$ avec $\forall i \in \mathbb{N},\ m_i > m_{i+1}$.

strictement inférieure au nœud expansé, c'est-à-dire si on a

$$\frac{b_s \in \mathsf{Tree}(\mathcal{N}, \mathsf{Subtree}(a) \setminus \{a\})}{\dfrac{b \in_{\pi_b} a \qquad b \rightsquigarrow_\beta \mathsf{join}(b_s)}{a \rightsquigarrow a[\pi \leftarrow \mathsf{join}(b_s)] = a'}}$$

où les nœuds de b_s sont strictement inférieurs au nœud b :

$$\forall \pi,\ \mathsf{node}(c, \_) \in_\pi b_s \implies m_{\mathcal{N}}(c) < m_{\mathcal{N}}(\mathsf{constr}(b))$$

Lemme 4. L'expansion d'expressions de type annotées est une expansion mesurée.

Démonstration. Une expansion de tête dans un type annoté est de l'une des formes

$$\frac{t := \lambda(\alpha_i)_i.\ \tau' \qquad t \notin l}{(\tau_i)_i\ t \mathbin{@} l \rightsquigarrow_\beta \tau'[(\alpha_i \leftarrow \tau_i)_i]^{@l,t}} \qquad \frac{t := \lambda(\alpha_i)_i.\ \tau' \qquad t \in l}{(\tau_i)_i\ t \mathbin{@} l \rightsquigarrow_\beta \texttt{Cycle}}$$

Comme mesure d'un nœud $t \mathbin{@} l$, on prend tout simplement la trace l, ordonnée par anti-inclusion : $l \leq l'$ si et seulement si $l \supseteq l'$. Par construction, nos traces ne peuvent mentionner qu'au plus une fois chaque constructeur de type, et le nombre de constructeurs de types déclarés dans le programme programme est fini ; il existe donc une taille maximale de traces possibles (le nombre de définitions de type), et l'ordre sur les traces est bien fondé. Enfin, lors de l'expansion d'un nœud $\tau \mathbin{@} l$, les nouveaux nœuds viennent de la substitution $\tau'[(\alpha_i \leftarrow \tau_i)_i]^{@l,t}$, et ils sont donc annotés par une trace strictement plus petite l, t.

Pour les réductions $\tau \rightsquigarrow_\beta \texttt{Cycle}$, on ajoute aussi `Cycle` à cet ensemble de mesures, en étendant l'ordre avec $\texttt{Cycle} < l$ pour tout l ; il reste bien fondé. Pour résumer :

$$\begin{array}{lcl} m(\tau \mathbin{@} l) & := & l \\ m(\texttt{Cycle}) & := & \texttt{Cycle} \end{array} \qquad \frac{l \supseteq l'}{l \leq l'} \qquad \frac{}{\texttt{Cycle} < l}$$

□

Théorème 1 (Terminaison)**.** Si $\mathcal{N}$ est muni d'une mesure $m_{\mathcal{N}}$, les expansions mesurées sur $\mathsf{Tree}(\mathcal{N})$ sont fortement normalisantes : il n'existe pas de séquence infinie d'expansions mesurées.

Pour montrer la terminaison de l'expansion d'arbre, on utilise notre mesure $m_{\mathcal{N}}$ sur les nœuds pour construire une mesure sur les arbres, qui envoie les arbres sur un ensemble muni d'un ordre bien fondé, telle qu'une expansion fait décroire strictement la mesure d'un arbre.

Étant donné un ensemble $(S, <_S)$ muni d'un ordre bien fondé, on peut construire un ordre bien fondé $(<_{\mathcal{P}(S)})$ sur l'ensemble $\mathcal{P}(S)$ des ensembles d'éléments de S, et un ordre bien fondé $(<_{\mathcal{M}(S)})$ sur l'ensemble $\mathcal{M}(S)$ des multi-ensembles d'éléments de S (cet ordre est attribué à Dershowitz, Manna, Huet et Oppen).

— $S_1 <_{\mathcal{P}(S)} S_2$ si, pour tout $a \in S_1$, il existe $b \in S_2$ tel que $a <_S b$.

— $M_1 <_{\mathcal{M}(S)} M_2$ si M_1 est de la forme $M + N_1$ et M_2 est de la forme $M + N_2$ avec $\lfloor N_1 \rfloor_{\mathsf{set}} <_{\mathcal{P}(S)} \lfloor N_2 \rfloor_{\mathsf{set}}$.

À partir d'une mesure $m_{\mathcal{N}} : \mathcal{N} \to (O, <_O)$ sur les étiquettes des nœuds de nos arbres (où $(<_O)$ est bien fondé), on définit pour tout arbre $a \in \mathsf{Tree}(\mathcal{N})$ une mesure $m_{\mathsf{Node}(a)} : \mathsf{Subtree}(a) \to (\mathcal{M}(O), <_{\mathcal{M}(O)})$ sur les nœuds de a, en considérant pour chaque nœud le chemin entre ce nœud (inclus) et la racine de l'arbre, vu comme un multi-ensemble de nœuds.

On définit notre mesure $m_{\mathsf{Node}(a)}(b)$, pour chaque nœud $b \in \mathsf{Subtree}(a)$, comme la somme de la mesure du constructeur de b et d'une mesure $m_{\mathsf{Path}(a)}$ définie par induction sur $b \in_\pi a$.

$$m_{\mathsf{Node}(a)}(b) \quad := \quad \{\!\{m_{\mathcal{N}}(\mathsf{constr}(b))\}\!\} + m_{\mathsf{Path}(a)}(b \in_\pi a) \qquad m_{\mathsf{Path}(a)}\left(a \in_\emptyset a\right) := \emptyset$$

$$m_{\mathsf{Path}(\mathsf{node}(c,\,(a_i)_i))}\left(\frac{b \in_\pi a_i}{b \in_{\pi.i} \mathsf{node}(c,\,(a_i)_i)}\right) := \{\!\{m_{\mathcal{N}}(c)\}\!\} + m_{\mathsf{Path}(a_i)}\left(b \in_\pi a_i\right)$$

Notons que la mesure $m_{\mathsf{Path}(a)}$ (pas $m_{\mathsf{Node}(a)}$) est compatible avec la concaténation de chemins :

$$a_1 \in_{\pi_{12}} a_2 \quad \wedge \quad a_2 \in_{\pi_{23}} a_3$$

$$\implies \qquad m_{\mathsf{Path}(a_3)}(a_1 \in_{\pi_{12}.\pi_{23}} a_3) \quad = \quad m_{\mathsf{Path}(a_2)}(a_1 \in_{\pi_{12}} a_2) + m_{\mathsf{Path}(a_3)}(a_2 \in_{\pi_{23}} a_3)$$

À partir de cette mesure $m_{\mathsf{Node}(a)} : \mathsf{Subtree}(a) \to (\mathcal{M}(O), <_{\mathcal{M}(O)})$ on définit une mesure $m_{\mathsf{Tree}} : \mathsf{Tree}(\mathcal{N}) \to (\mathcal{M}(\mathcal{M}(O)), <_{\mathcal{M}(\mathcal{M}(O))})$ sur les arbres, en considérant pour chaque arbre le multi-ensemble des mesures de ses nœuds. Les arbres sont donc mesurés comme des multi-ensembles de multi-ensembles d'étiquettes munies d'un ordre bien fondé :

$$m_{\mathsf{Tree}}(a) = \{\!\{m_{\mathsf{Node}(a)}(b) \mid b \in \mathsf{Subtree}(a)\}\!\}$$

Il faut montrer que les expansions mesurées pour $m_{\mathcal{N}}$ font strictement décroitre m_{Tree}. Cette deuxième partie de la preuve, moins difficile que ce bon choix de mesure, est faite en Annexe A.1.

4.5 Complétude ? « Oui ! Pour l'instant. »

On sait maintenant que la relation de récriture sur les expressions de type annotées $\tau \mathbin{@} l$ de recherche de preuve de termine nécessairement. En particulier, pour toutes les définitions récursives dont l'expansion ne termine pas, notre algorithme renverra `Cycle` en un temps fini – le nombre d'expansions est même borné par le nombre de constructeurs de type définis dans l'environnement. De plus, pour tous les exemples non-cycliques que nous avons considérés jusque-là, l'algorithme calcule effectivement la forme de tête et ne s'arrête pas (à tort) avec le résultat `Cycle`. Existe-t-il des définitions pour lesquelles l'algorithme renvoie `Cycle`, alors que la séquence d'expansions aurait en fait terminé ?

La réponse à cette question dépend de l'expressivité du langage de déclarations de type. Dans le fragment que nous avons présenté, la réponse est *non* : notre détection de la terminaison est correcte *et* complète. Pour le langage OCaml tout entier, nous conjecturons que notre algorithme est toujours complet ; mais l'ajout de fonctionnalités plus avancées (types de sorte supérieure, etc.) pourrait perdre cette propriété. Sur le long terme, nous nous contentons d'affirmer que l'algorithme termine sur tous les exemples *utiles* que nous avons rencontré, et qu'il pourrait être à raffiner si des fonctionnalités avancées sont ajoutées au langages, qui permettent d'écrire des définitions utiles sur lequel il serait incomplet. Nous considérons donc le résultat de complétude, esquissé ci-dessous, comme une curiosité temporaire.

La complétude de notre algorithme peut sembler surprenante ; d'habitude dans notre domaine un algorithme est soit correct, soit complet, mais jamais les deux a la fois, car le problème sous-jacent est indécidable. En général le problème de la terminaison d'un λ-terme en présence de définitions récursives arbitraires est certainement indécidable. Mais les λ-termes correspondant aux définitions de type ML (voir Section 4.2) des propriétés spécifiques :

— La seule règle de réduction est la β-expansion des fonctions, tous les autres constructeurs de termes sont inertes/neutres.

— On est dans un fragment de *premier ordre* : les termes sont bien typés pour un système de types simples où les arguments d'une fonction ont toujours le type de base α, jamais un type de fonction. Par exemple, un type paramétré $(\alpha, \beta)\, t$ a toujours le type (la sorte) $\star \times \star \to \star$, notre grammaire ne permet pas de placer une variable α en position de constructeur de type.

Dans ce fragment, une β-expansion ne crée jamais de « nouveaux » radicaux (redexes) : les arguments d'une fonction ne sont jamais des λ-abstractions dont la substitution créerait un nouveau radical, les radicaux sont donc déjà présentes dans les arguments expansés ou dans le corps de la fonction.

En particulier, si on a un type $(\tau_i)_i\, t$ dont les arguments τ_i sont normalisants, mais qui n'est pas normalisant lui-même[5], alors la non-terminaison ne dépend pas des τ_i : le terme $(\alpha_i)_i\, t$ ne terminerait pas non plus. La source de non-terminaison (le prochain radical dans la séquence de réductions infinie) vient de la définition du constructeur de type t. Dans notre système annoté, si le type $(\tau_i)_i\, t$ a la trace l, alors le prochain radical aura la trace l, t, et ainsi de suite. On extraire de toute séquence de réduction infinie une sous-séquence où chaque radical a une trace plus grande que le précédent. En conséquence, notre critère dynamique de détection des boucles est complet, il interromprait toute séquence de réduction infinie dans ce langage.

Conclusion

Bien gérer les constructeurs emboîtés demande de résoudre un problème de terminaison en présence de définitions récursives étonnamment intéressant.

Remarque. Nous manquons de place pour parler d'autres aspects de ce travail, que nous avons donc laissés en appendices dans les pages qui suivent.

Acknowledgments We received good feedback on this work from the OCaml community, in particular the excellent comments of Stephen Dolan informed several aspects of our work. Stephen was also effective as a « proof assistant », finding flaws in our first three/four proof attempts. Irène Waldspurger suggested measuring nodes by their path to the root, the key ingredient missing from our previous proofs. Finally, the combative review of Adrien Guatto greatly improved the presentation.

Références

Nicolas Chataing and Gabriel Scherer. Constructor unboxing: modified proposal, 2021.

Simon Colin, Rodolphe Lepigre, and Gabriel Scherer. Unboxing Mutually Recursive Type Definitions in OCaml. In *JFLA 2019*, January 2019.

Torbjörn Granlund and contributors. Gmp, 1991.

Antoine Miné and Xavier Leroy. Zarith, 2012.

Jeremy Yallop. Rfc proposal: constructor unboxing, 2020.

5. Si on a un terme non-normalisant, il existe toujours un sous-terme de cette forme : soit le constructeur de tête a des arguments normalisants, soit l'un des arguments est non-normalisant et on raisonne récursivement sur celui-là.

A Haut niveau (grenier)

A.1 Terminaison

Preuve du théorème de terminaison (Théorème 1). Considérons une expansion mesurée $a \rightsquigarrow a'$, elle est nécessairement de la forme suivante, où un sous-arbre b à un chemin π_b dans a est remplacé par un sous-arbre $\mathsf{join}(b_s)$, où les nœuds de b_s sont de mesure strictement plus petite :

$$\frac{b_s \in \mathsf{Tree}(\mathcal{N}, \mathsf{Subtree}(a) \setminus \{a\}) \qquad \frac{b \in_{\pi_b} a \qquad b \rightsquigarrow_\beta \mathsf{join}(b_s)}{a \rightsquigarrow a[\pi_b \leftarrow \mathsf{join}(b_s)] = a'}}{\forall \pi, (\mathsf{node}(c, \_)) \in_\pi b_s \implies m_{\mathcal{N}}(c) < m_{\mathcal{N}}(\mathsf{constr}(b))}$$

On veut montrer que $m_{\mathsf{Tree}}(a) > m_{\mathsf{Tree}}(a')$.

Séparer les nœuds modifiés des nœuds inchangés On peut séparer les chemins π valides sur a et a' en deux ensembles : les chemins qui sont des extensions de π_b ($\pi \geq \pi_b$), qui désignent un sous-arbre du sous-arbre b (dans a) ou du sous-arbre $\mathsf{join}(b_s)$ (dans a'), et les chemins $\pi \not\geq \pi_b$ qui ne sont pas des extensions de π_b.

$$\begin{aligned}
& m_{\mathsf{Tree}}(a) \\
= \;& \{\!\{m_{\mathsf{Node}(a)}(b') \mid \exists b' \in \mathsf{Subtree}(a)\}\!\} \\
= \;& \{\!\{m_{\mathsf{Node}(a)}(b') \mid \exists b' \exists \pi,\ b' \in_\pi a\}\!\} \\
= \;& \{\!\{m_{\mathsf{Node}(a)}(b') \mid \exists b' \exists \pi \geq \pi_b,\ b' \in_\pi a\}\!\} + \{\!\{m_{\mathsf{Node}(a)}(b') \mid \exists b' \exists \pi \not\geq \pi_b,\ b' \in_\pi a\}\!\}
\end{aligned}$$

$$\begin{aligned}
& m_{\mathsf{Tree}}(a') \\
= \;& \{\!\{m_{\mathsf{Node}(a')}(b') \mid \exists b' \exists \pi \geq \pi_b,\ b' \in_\pi a'\}\!\} + \{\!\{m_{\mathsf{Node}(a')}(b') \mid \exists b' \exists \pi \not\geq \pi_b,\ b' \in_\pi a'\}\!\}
\end{aligned}$$

La mesure $m_{\mathsf{Node}(a)}(b')$ d'un sous-arbre b' de a ne dépend que des constructeurs sur le chemin entre la racine a et le sous-arbre b' (inclus). En particulier, si b' est à un chemin $\pi \not\geq \pi_b$ qui n'étend pas π_b, il est aussi présent au même chemin comme sous-arbre de a', où il a la même mesure, puisque les constructeurs de ces arbres diffèrent seulement sous π_b :

$$\{\!\{m_{\mathsf{Node}(a)}(b') \mid \exists b' \exists \pi \not\geq \pi_b,\ b' \in_\pi a\}\!\} \quad = \quad \{\!\{m_{\mathsf{Node}(a')}(b') \mid \exists b' \exists \pi \not\geq \pi_b,\ b' \in_\pi a'\}\!\}$$

Et il nous reste à montrer, par définition de l'ordre sur les multi-ensembles, la partie $\lfloor N_1 \rfloor_{\mathsf{set}} > \lfloor N_2 \rfloor_{\mathsf{set}}$:

$$\{m_{\mathsf{Node}(a)}(b') \mid \exists b' \exists \pi \geq \pi_b,\ b' \in_\pi a\} \quad >_{\mathcal{P}(\mathcal{M}(O))} \quad \{m_{\mathsf{Node}(a')}(b') \mid \exists b' \exists \pi \geq \pi_b,\ b' \in_\pi a'\}$$

Classification d'un chemin dans $\mathsf{join}(b_s)$ Pour chaque sous-arbre sous π_b dans a', donc dans $\mathsf{join}(b_s)$, il faut montrer qu'il existe un sous-arbre sous π_b dans a, donc dans b, de mesure strictement plus grande.

Remarquons (et prouvons) qu'un sous-arbre b' de a' a un chemin de la forme $\pi_s.\pi_n.\pi_b$, avec :

- le chemin π_b entre b et la racine a,
- Un chemin π_n ($n$ouveaux nœuds) de zéro, un ou plusieurs « nouveaux » nœuds de l'expansion,
- un chemin π_s ($s$ous-arbre), éventuellement vide, dans une copie d'un sous-arbre strict de b.

Quand π_s termine bien dans un sous-arbre copié de b, on définit de plus un chemin π_v ($v$ieux nœuds) qui va de b au premier vieux nœud recopié. (Sinon π_v est choisi vide.)

Par exemple, dans la Figure 4.4.2, certains chemins vers des nœuds de l'expansion de c sont les suivants – pour plus de lisibilité, on écrit les chemins en indiquant le constructeur du nœud plutôt que son indice :

— $[a, c']$, qui est de la forme $\pi_b = [a]$, $\pi_n = [c']$, $\pi_s = []$ avec $\pi_v = []$.
— $[a, c', c'', f]$, qui est de la forme $\pi_b = [a]$, $\pi_n = [c', c'']$, $\pi_s = [f]$ avec $\pi_v = [e]$.

La présence de π_b est par hypothèse, on a supposé $\pi \geq \pi_b$ donc $\pi = \pi'.\pi_b$ pour un certain π' tel que $b' \in_{\pi'} \mathsf{join}(b_s)$. Formellement, on définit comment découper π' en une paire (π_s, π_n) en définissant une fonction $\mathsf{split\_join}\,(b_s,\ b' \in_{\pi'} \mathsf{join}(b_s))$ par induction sur b_s et inversion sur la définition de $\mathsf{join}(b_s)$:

$$\mathsf{split\_join}\left(\mathsf{var}(x),\ \frac{x \in \mathsf{Tree}(\mathcal{N}, \mathcal{V})}{b' \in_{\pi''} \mathsf{join}(\mathsf{var}(x)) = x}\right) := (\pi'', \emptyset)$$

$$\mathsf{split\_join}\left(\mathsf{node}(c,\ \_),\ \frac{}{b' \in_{\emptyset} \mathsf{node}(c,\ \_) = b'}\right) := (\emptyset, \emptyset)$$

$$\mathsf{split\_join}\left(\mathsf{node}(c,\ (a_i)_i),\ \frac{b' \in_{\pi''} \mathsf{join}(a_i)}{b' \in_{\pi''.i} \mathsf{join}(\mathsf{node}(c,\ (a_i)_i))}\right) := (\pi''_s, \pi''_n.i)$$
$$\text{où} \quad (\pi''_s, \pi''_n) = \mathsf{split\_join}\,(a_i,\ b' \in_{\pi''} \mathsf{join}(a_i))$$

Par construction nous avons $\pi_s.\pi_n = \pi'$, donc $b' \in_{\pi_s.\pi_n} \mathsf{join}(b_s)$, et nous avons aussi $b' \in_{\pi_s} \mathsf{join}(b'_s)$ et $b'_s \in_{\pi_n} b_s$ pour un sous-arbre b'_s de b_s tel que :

(1) Ou bien b' n'est pas de la forme $\mathsf{var}(x)$: le chemin π' s'arrête dans b_s avant d'arriver à un nœud $\mathsf{var}(x)$. Dans ce cas b' est un « nouveau nœud » de b_s – en particulier, on sait que l'expansion a généré au moins un nouveau nœud, sinon on aurait $b_s = \mathsf{var}(\_)$. On a $\pi_s = \emptyset$ par définition de $\mathsf{split\_join}\,(\_,\ \_)$, et on définit en plus π_v comme $\emptyset$: il n'y a pas de chemin vers un ancien nœud correspondant dans b.

(2) Ou bien le chemin π' va jusqu'à un nœud $b'_s = \mathsf{var}(b'')$ de b_s : π_n est le chemin entre b'_s et b_s, et π' continue avec un chemin $b' \in_{\pi_s} b''$, éventuellement vide, jusqu'à un b'' sous-arbre strict de b. On définit alors π_v comme le chemin de b'' dans b, non-vide puisque b'' est un sous-arbre strict.

On va conclure la preuve en trouvant un sous-arbre de a sous π_b (donc dans b) dont la mesure est strictement supérieure à celle de notre sous-arbre b' de chemin $\pi_s.\pi_n.\pi_b$ dans a'.

Chemin supérieur Nous choisissons le sous-arbre de chemin $\pi_s.\pi_v.\pi_b$ dans a, que nous appelons b'_v.

Cas (1) Dans le cas (1) ci-dessus ($\pi_s = \pi_v = \emptyset$), b' est un nouveau nœud, et b'_v est le sous-arbre de chemin $\emptyset.\emptyset.\pi_b$, donc π_b : le nœud b'_v est exactement b. Il faut montrer qu'on a $m_{\mathsf{Node}(a)}(b) > m_{\mathsf{Node}(a')}(b')$.

$$\begin{aligned}
&m_{\mathsf{Node}(a)}(b'_v)\\
&= m_{\mathsf{Node}(a)}(b)\\
&= m_{\mathsf{Path}(a)}(\pi_b) + \{\!\{m_{\mathcal{N}}(\mathsf{constr}(b))\}\!\}
\end{aligned}$$

$$\begin{aligned}
&m_{\mathsf{Node}(a')}(b')\\
&= m_{\mathsf{Path}(a')}(\pi_n.\pi_b) + \{\!\{m_{\mathcal{N}}(\mathsf{constr}(b'))\}\!\}\\
&= m_{\mathsf{Path}(\mathsf{join}(b_s))}(\pi_n) + m_{\mathsf{Path}(a')}(\pi_b) + \{\!\{m_{\mathcal{N}}(\mathsf{constr}(b'))\}\!\}\\
&= m_{\mathsf{Path}(\mathsf{join}(b_s))}(\pi_n) + m_{\mathsf{Path}(a)}(\pi_b) + \{\!\{m_{\mathcal{N}}(\mathsf{constr}(b'))\}\!\}
\end{aligned}$$

L'expansion de a à a' est mesurée, donc on a $m_{\mathcal{N}}(\mathsf{constr}(b)) > m_{\mathcal{N}}(\mathsf{constr}(b'))$, qui conclut ce cas.

Cas (2) Dans le cas (2) ci-dessus, b' appartient à b'', un sous-arbre strict de b recopié par l'expansion selon b_s. π_s est le chemin de b' dans b'', et π_v le chemin (non-vide) de b'' dans b. On a :

$$\begin{array}{rcccccccc} m_{\mathsf{Node}(a)}(b'_v) & = & m_{\mathsf{Path}(a)}(\pi_b) & + & m_{\mathsf{Path}(b)}(\pi_v) & + & m_{\mathsf{Path}(b'')}(\pi_s) & + & m_{\mathcal{N}}(b'_v) \\ & & (=) & & & & (=) & & (=) \\ m_{\mathsf{Node}(a')}(b') & = & m_{\mathsf{Path}(a')}(\pi_b) & + & m_{\mathsf{Path}(b_s)}(\pi_n) & + & m_{\mathsf{Path}(b'')}(\pi_s) & + & m_{\mathcal{N}}(b') \end{array}$$

Pour conclure il faut montrer $m_{\mathsf{Path}(b)}(\pi_v) > m_{\mathsf{Path}(b_s)}(\pi_n)$. Cela est vrai car π_v est non-vide, donc $m_{\mathsf{Path}(b)}(\pi_v)$ est une somme qui contient en particulier la mesure de sa racine $m_{\mathcal{N}}(b)$, qui est strictement supérieure à la mesure de tous les nœuds de b_s puisque l'expansion est mesurée. □

B Bas niveau

B.1 Représentation des autres valeurs (que les types sommes)

Nous avons décrit dans l'introduction la façon dont les types sommes sont représentés soit comme des valeurs immédiates, soit par un *bloc* avec des méta-données en en-tête. Plus généralement, toutes les valeurs OCaml sont représentées ainsi. Par exemple, `int`, `bool` et `char` sont représentés comme des valeurs immédiates, alors que `float`, `string`, `int array` ou bien `exn` (le type des exceptions) sont représentés par des blocs.

Plus précisément :

— Les types « produits » (n-uplets (`int * bool * float`), les enregistrements et les tableaux[6]) portent l'étiquette 0, comme le premier constructeur d'un type somme.

— Les autres types de valeurs portent une étiquette spécifique, qui sert au ramasse-miette et aux fonctions de l'environnement d'exécution (comparaison polymorphe, sérialisation etc.) pour les distinguer. Par exemple, les `string` portent l'étiquette `Obj.string_tag = 252`, les objets et les types sommes extensibles (dont `exn`) portent l'étiquette `Obj.object_tag = 248`, et les types « étrangers (custom) », implémentés en C (dont `int32`, `int64`, `nativeint`), portent l'étiquette `Obj.custom_tag = 255`. Ces tags particuliers sont autour de 250, car l'étiquette d'une valeur doit tenir dans un octet, et une plage d'étiquettes entre 0 et `Obj.last_non_constant_constructor_tag = 245` est réservée pour les constructeurs non-constants et les types produits.

Ainsi, la définition suivante est acceptable

```
type tree =
  | Concat of tree * tree
  | Leaf of string [@unboxed]
```

car déboîter le constructeur `Leaf` introduit des valeurs d'étiquette `Obj.string_tag = 252`, alors que les valeurs du constructeur `Concat` portent l'étiquette 0. Mais la définition suivante serait incorrecte :

```
type tree =
  | Concat of tree * tree
  | Leaf of string array [@unboxed]
Error: This declaration is invalid, some [@unboxed] annotations introduce
       overlapping representations.
```

6. Pour les spécialistes : les tableaux de flottants peuvent aussi porter l'étiquette `Obj.double_array_tag = 254`. C'est toujours l'unique tag du type `FloatArray.t`, et c'est aussi un tag possible pour `'a array` à moins que l'option `-no-flat-float-array` ne soit utilisée.

car les `string array` portent l'étiquette 0.

B.2 Représentation de bas niveau des têtes

Dans notre implémentation, nous suivons cette représentation des valeurs en définissant la tête d'une valeur $\texttt{head}(v)$ comme un élément de $\{\texttt{Imm}, \texttt{Block}\} \times \mathbb{Z}_{\texttt{int}}$ (où $\mathbb{Z}_{\texttt{int}}$ est l'ensemble des entiers OCaml de type `int`) ainsi :

— La tête d'une valeur immédiate n est la paire $(\texttt{Imm}, n)$

— La tête d'un bloc de tag t est la paire $(\texttt{Block}, t)$.

De façon équivalente, nous pouvons décrire les têtes par un type de donnée ML (utilisé de nouveau comme un méta-langage) :

```
type imm = Imm of int [@unboxed]
type tag = Tag of int [@unboxed]
type head =
| Imm of imm
| Block of tag
```

Les ensembles de têtes qui approximent les types OCaml peuvent être représentés facilement. Dans chaque domaine (les valeurs immédiates ou les étiquettes de blocs), on représente un ensemble par soit un ensemble fini de cas possibles, soit un élément $\top$ représentant l'ensemble des valeurs possibles du domaine. Nos formes de tête de bas niveau sont décrites ainsi dans notre méta-langage :

```
type 'a shape =
  | Set of 'a list
  | Any
type head_shape = {
  imm: imm shape;
  tag: tag shape;
}
```

Pour les constructeurs emboîtés C, la représentation de bas niveau est déterminée par la présence ou non d'un argument, et leur ordre dans la déclaration de type de donnée : les constructeurs constants ont la tête `Imm n` où n est leur rang (à partir de 0) parmi les constructeurs constants, et pareillement les constructeurs non constants ont la tête `Tag n`.

Pour les autres types, on utilise la forme de tête qui approxime la représentation des valeurs par le compilateur OCaml. Voici quelques exemples de formes de têtes :

— `int` : `{imm = Any; tag = Set []}`

— `bool` : `{imm = Set [Imm 0; Imm 1]; tag = Set []}`

— `int64` : `{imm = Set []; tag = Set [Obj.custom_tag]}`

— `exn` : `{imm = Set []; tag = Set [Obj.object_tag]}`

— `('a * 'b)` : `{imm = Set []; tag = [Tag 0]}`

— `('a option)`, `('a list)` : `{imm = Set [Imm 0]; tag = Set [Tag 0]}`

— `('a -> 'b)` : `{imm = Set []; tag = Set [Tag Obj.closure_tag; Tag Obj.infix_tag]}`[7]

— `('a array)` : `{imm = Set []; tag = Set [Tag 0; Tag Obj.double_array_tag]}`[8]

— `('a Lazy.t)` : `{imm = Any; tag = Any}`[9]

7. `Obj.infix_tag` signale certaines fonctions d'un bloc de fonctions mutuellement récursives.

8. Remarque pour les spécialistes : pour les tableaux, `Obj.double_array_tag` est présent ou non selon que l'option `-no-flat-float-array` est activée. En l'absence de tableaux de flottants plats, on peut définir `type _ arr = Array : 'a array -> 'a arr [@unboxed] | Float : Floatarray.t -> float arr [@unboxed]`.

9. Remarque pour les spécialistes : l'implémentation de référence OCaml utilise une optimisation de raccourci pour les valeurs paresseuses : si on veut construire un glaçon à partir d'une valeur déjà calculée (plutôt qu'un

À l'exception des cas `'a option` et `'a list`, les exemples ci-dessus correspondent tous à des types primitifs du langage, qui déterminent l'ensemble les types primitifs $\widehat{\texttt{t}}$ dans nos jugements de calcul de la forme de tête, et leur représentation de bas niveau.

Ces informations suffisent à calculer la représentation de bas niveau d'une tête h et d'une forme de tête H, et donc de rejeter les définitions de types dont les têtes de bas niveau contiennent des doublons.

B.3 Compilation du filtrage par motif

Le compilateur OCaml stocke, dans ses environnements, des méta-données sur les déclarations de types et leurs constructeurs ; en particulier il stocke déjà, pour les constructeurs emboîtés, leur représentation (valeur immédiate ou étiquette de bloc). Nous étendons ces méta-données pour stocker, pour chaque constructeur déboîté, la forme de tête H de son paramètre de type.

Pour comprendre notre schéma de compilation, il faut connaître deux constructions déjà utilisées par la traduction du filtrage de motif en OCaml, dans la représentation intermédiaire « Lambda » :

— Une construction `switch` sur une valeur OCaml, dont les cas correspondent soit à une valeur immédiate soit à l'étiquette d'un bloc, optionnellement un « cas par défaut » pour les entrées n'ayant pas de cas spécifique.

```
type formula = True | False | And of formula * formula | Or of formula * formula
let rec eval f =
  match f with
  | True -> true
  | False -> false
  | And (f1, f2) -> eval f1 && eval f2
  | Or (f1, f2) -> eval f1 || eval f2

(* ocamlc -dlambda -dno-unique-ids test.ml *)
(switch* f
 case int 0: 1
 case int 1: 0
 case tag 0: (&& (apply eval (field 0 f)) (apply eval (field 1 f)))
 case tag 1: (|| (apply eval (field 0 f)) (apply eval (field 1 f))))
```

(`switch*` plutôt que `switch` indique l'absence de cas par défaut : la liste de cas est exhaustive pour toutes les entrées possibles.)

— Une primitive `isint` qui teste si une valeur est une valeur immédiate ou un bloc. Elle est utilisé dans le cas où au moins l'un des « domaines » du filtrage ne contient qu'un seul cas (et pas de cas par défaut).

```
let rec size f = match f with
  | True | False -> 1
  | And (f1, f2) | Or (f1, f2) -> 1 + size f1 + size f2

(* ocamlc -dlambda -dno-unique-ids test.ml *)
(if (isint f)
  1
  (+ (+ 1 (apply size (field 0 f))) (apply size (field 1 f))))
```

calcul restant à effectuer), il peut être représenté directement par cette valeur sans l'emboîter. Cela est possible à n'importe quelle type, donc les valeurs paresseuses peuvent contenir n'importe quelle tête.

Le compilateur de filtrage par motif génère ces deux constructions à partir de listes de clauses simplifiées, indicées chacune par une constructeur du type filtré (sans nommer ses variables, etc.).

En présence seulement de constructeurs emboîtés, il suffit au compilateur de regarder la représentation (`Imm n` ou `Tag n`). Si le domaine des constructeurs non-constants ne contient qu'un seul cas, il génère un `if (isint arg) ...`, avec un arbre de tests conditionnels bien choisis pour le domaine constant. Sinon, il génère une instruction `switch`, qui est préservée par le compilateur vers machine virtuelle (bytecode) (la machine virtuelle a une instruction `switch`) et transformée ultérieurement en tests bien choisis par le compilateur natif.

Quand on ajoute des constructeurs déboîtés, il faut gérer le fait qu'un `switch` peut contenir des cas correspondant à un constructeur déboîté, dont la forme de tête (stockée dans l'environnement et donc disponible pendant la compilation) indique plusieurs valeurs immédiates ou tags possibles. Par exemple, on pourrait se retrouver avec un `switch` ressemblant au suivant, en présence d'un constructeur constant (`Tag 0`), d'un constructeur déboîté de forme de tête `{imm = Set [Imm 0]; tag = Set [Tag 252]}`, et d'autres cas traités par une clause par défaut :

```
(switch* f
 case tag 0: foo
 case unboxed (int 0 | tag 252): bar
 default: foobar)
```

Notre stratégie de compilation, très simple, consiste à expanser ces cas en plusieurs cas, un par valeur de tête possible :

```
(switch f
 case tag 0: foo
 case int 0: bar
 case tag 252: bar
 default: foobar)
```

Cette transformation duplique des « actions » (les expressions du côté droit), mais le compilateur de filtrage par motif a déjà une machinerie pour garder trace de ces duplications et insérer des constructions de partage.

Enfin, un constructeur déboîté peut contenir la tête `Any` pour l'un des domaines ou les deux. Notre exemple précédent, avec la forme de tête `{imm = Any; tag = Set [Tag 252]}`, donnerait :

```
(switch* f
 case tag 0: foo
 case unboxed (int any | tag 252): bar
 default: foobar)
```

Dans ce cas, nous ne pouvons pas générer seulement la construction `switch`, qui ne permet pas de donner une condition « pour toutes les valeurs immédiates » ou « pour toutes les étiquettes de bloc ». Nous utilisons alors la construction `isint` pour traiter différemment les deux domaines :

```
(if (isint f)
  bar
  (switch* f
   case tag 0: foo
   case tag 252: bar
   default: foobar))
```

Inconvénient : motifs épars Une construction `switch` avec un cas `tag 0` et un cas `tag 252` génère en fait du code un peu décevant sur la machine virtuelle (bytecode), on obtient une instruction avec une table de sauts de 252 octets. Idéalement il faudrait repérer ces cas de tags très élevés et les gérer avec des conditionnelles avant le `switch`. En pratique, la machine virtuelle a maintenant peu d'utilisateurs, donc ce travail n'est pas une priorité.

Inconvénient : tests répétés Notre stratégie, très simple, génère parfois du code décevant, quand un constructeur déboîté a lui-même comme argument un type somme sur lequel on fait de nouveau un filtrage. Par exemple :

```
type t = Int of int | String of string
type u = Unit | Const of t [@unboxed]

let to_string u =
  match u with
  | Unit -> "()"
  | Const (Int n) -> string_of_int n
  | Const (String s) -> s
```

Ce programme va générer deux `switch` imbriqué, alors qu'un seul suffirait :

```
(switch* u
  int 0: "Unit"
  tag 0 | tag 1:
    (switch u
      tag 0: (apply string_of_int (field u 0))
      tag 1: (field u 1)))
```

Régler ce problème demande des changements plus invasifs au compilateur de filtrage par motif, sur lesquels nous n'avons pour l'instant pas travaillé. Les cas d'usage principaux pour le déboîtement de constructeurs que nous connaissons déboîtent un paramètre à un type primitif (`int`, `string`, etc.), et pas un type somme, donc ne rencontrent pas ce problème.

C Travaux futurs

C.1 Portabilité à d'autres implémentations

Un défaut de ce travail est que la décision d'accepter ou rejeter des déclarations de constructeurs déboîtés dépend de détail bas-niveau de l'implémentation du langage, qui n'ont pas de raison d'être communs à différentes implémentations.

Nous aimerions donner une sémantique de plus haut niveau, c'est-à-dire choisir un critère de non-confusion pour les formes de tête de haut niveau qui soit plus simple, moins spécifique, donc portable à plus d'implémentations variées du langage.

Mais toute simplification a pour effet d'effacer des distinctions entre types, et donc d'augmenter les approximations et d'accepter moins d'annotations de déboîtement ; elle a donc un impact en performance pour les utilisateurs de l'implémentation principale.

Nous proposons pour l'instant deux approches pour atténuer ce problème de conception :

— Par défaut, utiliser une représentation bas-niveau des formes de tête un peu moins fine, qui utilise un « plus petit dénominateur commun » entre l'implémentation de référence de OCaml et `js_of_ocaml`, l'implémentation utilisée dont la représentation des valeurs est la plus différente.

— Envisager des annotations qui indiquent explicitement qu'elles ne seront honorées que par certaines annotations, par exemple une annotation `[@unboxed native]` qui ne prend

effet qu'avec l'implémentation de référence.

La différence de représentation principale de `js_of_ocaml` se situe au niveau des types numériques : `int`, `float`, `nativeint` et `int32` ont tous la même représentation sous-jacente dans cette implémentation, ainsi que les constructeurs constants.

Pour implémenter une notion de forme de tête moins fine, qui empêche les confusions entre les types ayant la même représentation avec `js_of_ocaml`, une approche simple est d'ajouter un champ à nos formes de tête, qui garde trace de si la tête inclus un de ces types numériques :

```
type head_shape = {
  imm: imm shape;
  tag: tag shape;
  numeric: bool;
}
```

Deux formes de tête sont en conflit (ne sont pas disjointes) si elles sont en conflit selon la définition habituelle, ou si le champ `numeric` est vrai des deux côtés.

C.2 Types abstraits

Un type abstrait présent dans une implémentation (pas une interface) de module est souvent destiné à être peuplé par du code écrit en C, ou dans un autre langage utilisant la IFE (FFI) OCaml. Nous pourrions laisser l'utilisateur indiquer la forme de tête de ces types abstraits ; il faudrait lui faire confiance (le compilateur ne va pas inspecter le code C pour vérifier), mais de nombreux aspects de l'interface IFE, comme par exemple le type des primitives `external`, reposent déjà sur le fait de faire confiance à la description de l'interaction côté OCaml.

C.3 Signatures, paramètres, modularité

Si un type abstrait peut être annoté avec une forme de tête, cette construction pourrait aussi être utilisée dans les signatures de modules ; il faudra alors savoir vérifier qu'une définition concrète du type vérifie bien la forme de tête utilisée dans la signature.

Enfin, nous pourrions envisager d'annoter un paramètre de type α_i d'une définition pour lui donner une forme de tête plus précise que $\top$, en interdisant les instances du type ne respectant pas cette contrainte.

Pour être pleinement modulaire, il faudrait aussi concevoir un langage pour décrire, pour un type abstrait paramétré $\alpha\, t$, la forme de tête des instances $\tau\, t$ en fonction de la forme de tête du paramètre τ – peut-être une extension du langage des formes de tête avec des variables formelles.

C.4 Cycles bénins

Il y a une différence entre les deux déclarations cycliques suivantes, rejetées par notre critère de terminaison qui échoue avec le résultat `Cycle` :

```
type bad_cycle = Foo | Loop of bad_cycle [@unboxed]

type meh_cycle = Loop' of meh_cycle [@unboxed]
```

Dans le premier cas, `bad_cycle`, autoriser la déclaration de type aboutirait vraiment à des confusions entre `Foo`, `Loop Foo`, `Loop (Loop Foo)`, etc. : elle doit être rejetée à la fois parce qu'elle est cyclique et parce qu'elle est incorrecte.

Dans le second cas, si la définition était acceptée, elle donnerait un type vide : on n'a pas de constructeur non-récursif utilisable comme cas de base, donc aucune valeur de ce type ne peut

être construite [10]. On peut parler d'un cycle « bénin », qui pourrait être accepté sans danger.

Stephen Dolan nous a fait remarquer que certains cas de définitions acceptées peuvent se transformer en cycles bénins quand des définitions de type abstraites sont révélées.

```
type 'a foo
type weird = Loop'' of weird foo [@unboxed]
```

Ce type est accepté par notre analyse, puisqu'on suppose la forme de tête $\top$ pour `'a foo` et, par extension, pour `weird`. Mais si on apprenait ensuite que `'a foo` était en fait définit par `type 'a foo = 'a`, on se rendrait compte qu'on a accepté une définition de cycle bénin.

Nous souhaitons préserver la propriété désirable que révéler la définition d'un type abstrait préserve la validité du programme. Nous devons donc modifier notre prototype pour accepter les cycles bénins.

Il n'est pas tout à fait évident de savoir comment distinguer les deux types de cycles dans notre algorithme. Une approche un peu naïve, et relativement simple à implémenter, consiste à autoriser l'expansion deux fois d'un même constructeur de types dans une branche de calcul (et d'arrêter si une troisième expansion est essayée), au lieu d'échouer avec `Cycle` à la deuxième expansion. Si le cycle n'est pas bénin, les autres formes de tête présentes dans le cycle vont être présentes en deux exemplaires (une par extension), donc créer des confusions et faire rejeter la déclaration.

C.5 Séparabilité

OCaml accepte déjà les déclarations de type à constructeur unique avec l'annotation `[@@unboxed]`. En raison de l'optimisation de tableaux de flottants (donc hors du mode `-no-flat-float-array`), cela demande une analyse spécifique pour garantir que les types déboîtés restent « séparables », c'est-à-dire qu'ils contiennent, dans l'ensemble de leurs valeurs, soit uniquement des nombres flottants soit aucun nombre flottant. (Pour plus de détails, voir Colin, Lepigre, and Scherer (2019).)

Avec un type mono-constructeur, la seule façon de casser la séparabilité est d'utiliser un type abstrait de GADT :

```
type any = Any : 'a -> any [@@unboxed]
Error: This type cannot be unboxed because
       it might contain both float and non-float values,
       depending on the instantiation of the existential variable 'a.
       You should annotate it with [@@ocaml.boxed].
```

Le déboîtement de constructeur permet de créer des types non-séparables sans utiliser de types existentiels :

```
type non_separable = Int of int [@unboxed] | Float of float [@unboxed]
```

Il faut adapter notre travail pour rejeter ces définitions et continuer à garantir la séparabilité. Nous proposons de le faire en ajoutant un booléen à notre définition de la forme de tête, qui trace la séparabilité de la forme de tête calculée :

```
type head_shape = {
  imm: imm shape;
  tag: tag shape;
  separable: bool;
}
```

10. Remarque : `let rec x = Loop' x in x` serait accepté si `Loop'` n'était pas un constructeur déboîté, mais est rejeté quand `Loop'` est marqué `[@unboxed]`.

Un type existentiel de GADT n'est pas séparable, et l'union d'une forme de tête autorisant les flottants avec une forme de tête autorisant autre-chose donne aussi une forme de tête non-séparable. Si la forme de tête de bas niveau calculée pour une définition contient `separable = false`, il faut rejeter la définition.

Remarque. Avant cette extension, on pouvait donner une sémantique à une forme de tête comme un ensemble de valeurs OCaml. Par exemple la sémantique de `{imm = Set []; tag = Set [Tag 0]}` était l'ensemble des blocs dont le tag est 0. Avec cette extension, on trace une propriété `separable: bool` relationnelle, qui parle des ensembles de valeur OCaml et non pas propriété d'une valeur en isolation.

On doit donc interpréter une forme de tête comme un *ensemble d'ensembles* de valeurs OCaml : `{imm = Set []; tag = Set [Tag 0]}` décrit l'ensemble des ensembles qui (1) contiennent uniquement des blocs de tag 0, et (2) sont séparables.

Remarque. Nous espérons que cette analyse dynamique (par dépliement de définitions) de la séparabilité des types pourrait remplacer entièrement l'analyse statique effectuée actuellement Colin, Lepigre, and Scherer (2019).

C.6 Arité

Notre notion de tête de bas niveau ne prend en compte que l'étiquette des blocs, mais on pourrait aussi prendre en compte son arité : deux constructeurs de même étiquette mais d'arité différente sont efficacement distinguables à l'exécution.

Par exemple, le programme suivant est rejeté avec notre présentation usuelle, et serait accepté si l'arité était ajoutée à nos têtes :

```
type 'a pair = 'a * 'a
type 'a triple = 'a * 'a * 'a

type 'a foo =
  | Pair of 'a pair [@unboxed]
  | Triple of 'a triple [@unboxed]
```

C.7 GADTs

On peut calculer plus finement la forme de tête d'un GADT. Considérons par exemple le type suivant :

```
type _ t =
  | Int : int -> int t [@unboxed]
  | Bool : bool -> bool t [@unboxed]
```

Avec notre approche actuelle, cette déclaration est rejetée ; notre analyse considère qu'il y a une confusion entre, par exemple, `Int 0` et `Bool false`, qui ont la même représentation. Mais, les types `int` et `bool` étant incompatibles, ces deux valeurs ont des types différents et ne peuvent pas être confondues l'une avec l'autre.

Nous pouvons donc raffiner notre calcul des formes de tête d'une expression de type $\tau\ t$, quand $\_\ t$ est un GADT. Nous proposons de le calculer ainsi :

1. Éliminer les constructeurs de t dont le type de retour est incompatible avec τ.
   Par exemple, si on cherche la forme de tête de `int t`, on ne garde que le constructeur `Int`, mais pour `'a t` ou `foo t` où `foo` est un type abstrait on garde les deux constructeurs.
2. Grouper les constructeurs restants en « composantes connexes », des ensemble de constructeurs dont les types de retour sont compatibles entre eux.

Un constructeur peut apparaître dans plusieurs composantes. Par exemple, si on rajoutait un constructeur `Id : 'a -> 'a t [@unboxed]` à notre exemple, les composantes connexes seraient `Int, Id` et `Bool, Id`.

3. Calculer la forme de tête de chaque composante connexe.
4. Renvoyer l'union (non-disjointe) des formes de tête obtenues.

Nous pensons que ce raffinement est important en pratique pour les utilisateurs de GADTs. Des types comme `_ t` ci-dessus sont souvent utilisés pour représenter les valeurs d'un mini-langage bien typé, et le fait de pouvoir déboîter toutes ces valeurs peut avoir un impact important sur les performances d'un évaluateur.

Une idée compliquée venant rarement seule, nous n'avons pas été si surpris de découvrir que ce raffinement a un impact sur la stratégie de détection des cycles, qui devient incomplète. Considérons par exemple le GADT suivant :

```
type _ t =
| A : int t -> bool t [@unboxed]
| B : int t
```

Avec notre jugement actuel, calculer la forme de tête de `bool t` demanderait de calculer la forme de tête de `int t`, et échouerait alors avec une erreur `Cycle`. Mais si nous avions accepté d'expanser une deuxième fois le constructeur `t`, nous l'aurions fait à une instance `int t` où seul le constructeur `B` est possible, et nous aurions pu terminer correctement notre calcul.

Notre vérification dynamique de terminaison repose sur l'hypothèse que si un constructeur apparaît dans son expansion, on a nécessairement une boucle infinie d'expansion – indépendamment des instances des paramètres du constructeur. Ce n'est plus vrai avec les GADTs, si nous utilisons ce critère raffiné où les instances des paramètres influent sur l'expansion de la définition.

Heureusement, il est relativement simple de restaurer la complétude dans ce cas. Pour les GADT, au lieu de stocker seulement le constructeur de type, ici `t`, dans la trace d'expansion, nous proposons de stocker le constructeur de type *et* les constructeurs qui ont été sélectionné, car leurs types sont compatibles avec l'instance spécifique `foo t` que l'on a expansé. Si `t` apparaît de nouveau en tête de l'expression de type à une instance `bar t`, on calcule de nouveau l'ensemble des constructeurs compatibles, et on échoue avec `Cycle` seulement si `t` existe déjà avec cet ensemble dans la trace. Ce critère préserve la terminaison, car il y a un nombre fini de sous-ensembles de constructeurs possibles pour `t`, et nous pensons qu'il restaure notre résultat de complétude pour les GADTs.