Généralisation de la Récursion terminale modulo constructeur

Stage 3-6 mois, niveau M1 à M2 Recherche.

Présentation d'OCamlPro :

OCamlPro a été fondé en 2011 par d'anciens chercheurs d'Inria avec pour mission d'aider les utilisateurs industriels à adopter OCaml, le langage de programmation de pointe. Nous concevons et mettons en œuvre des logiciels à forte valeur ajoutée pour nos clients et nous avons une longue expérience dans le développement et la maintenance d'outils open source pour OCaml.

Présentation d'OCaml :

OCaml est un langage fonctionnel de la famille des ML qui implémente la notion d'appel terminal. C'est-à-dire que si la dernière chose que fait une fonction avant de retourner est un appel de fonction, alors le nettoyage de la pile peut être effectué avant l'appel, évitant ainsi de faire croître la pile. C'est intéressant pour des questions de performance, mais surtout pour éviter d'atteindre les limites de la pile.
Par exemple pour une fonction telle que List.fold_left

```
let rec fold_left f accu l =
  match l with
[] -> accu
| a :: l ->
let accu = f accu a in
   fold_left f accu l
```

l'appel récusif à fold_left vérifie bien les conditions pour être terminal. La fonction fold_left peut donc être appelée sur des listes de longueur arbitraire sans risquer de débordement. Ce n'est pas le cas de List.map par contre qui sera donc limité à des list pas trop longues.

```
let rec map f = function
   [] -> []
| a :: l ->
let r = f a in
let tail = map f l in r ::
   :: tail
```

Une solution possible est la réécriture de fonctions critiques sous des formes telles qu'elles ne consommeront qu'une quantité finie de pile. C'est toujours faisable, mais le code résultant est souvent moins bon que la forme 'naive'. Une généralisation de la gestion des appels terminaux, la TRMC pour Tail Recursion Modulo Constructor, autorise à faire des allocations de 'constructeurs' après le dernier appel. La fonction List.map vérifie bien ce critère (le constructeur est : :). La TRMC est implémentée comme une transformation de code rendant l'appel effectivement terminal.

Cette transformation utilise des construction auquel le programmeur n'a pas accès (pour des raisons de sûreté de typage). Voir https://github.com/ocaml/ocaml/pull/9760 pour plus de détails. Mais il semble que cela puisse encore être généralisé. Par exemple, une transformation similaire est possible sur une fonction telle que :

```
type ('a, 'b) either = Left of 'a | Right of 'b

 let rec partition_map p xs =
  match xs with
| [] -> ([], [])
| x : : xs ->
let e = p x in
let (left, right) = partition_map p xs in
   match e with
| Left v ->
(v : : left, right)
   | Right v ->
    (left, v : : right)
```

Sujet de stage :

L'objectif de ce stage est d'étudier et implémenter cette généralisation. Le déroulement du stage suivrait probablement ce plan :

- Comprendre et définir dans quel cas une telle généralisation peut s'appliquer.
- Définir des critère compréhensibles par un programmeur pour savoir si cela peut s'appliquer à une fonction particulière.
- Implémenter un prototype de transformation
- Formaliser la transformation.
- Éventuellement étudier l'impact sur les performances des fonctions transformées

Le stage se déroulera au sein de l'équipe 'flambda' chez OCamlPro qui développe des optimisations sur le langage OCaml.