Notes: Space cost analysis using sized types

Introduction

Programmation embarquée

On veut pouvoir utiliser des langages de haut niveau dans les systèmes embarqués pour des raison évidentes (productivité accrue par rapport à C ou l'assembleur, etc). Mais cela vient avec une perte de prédictabilité quand à la consommation de ressource. Les tests et le profilage ne suffisent pas à *garantir* le bon comportement des programmes de haut niveau dans les contextes embarqués: on y cherche des performances *prévisibles* et *garanties* plutôt meilleures à tout prix.

On prendra soin de distinguer programmation *embarquée*, dans laquelle le programme doit jouer avec des ressources limités génériques, et la programmation *en temps réel*, dans laquelle le temps de latence est à borner. C'est cette première qui nous intéresse ici. Dans les systèmes embarqués traditionnels, on garantie la terminaison et la correction en excluant les constructions syntaxique les plus dangereuses (la récursion, l'allocation dynamique, les boucles sans bornes triviales…). Mais le développement de la programmation de haut niveau (*Java*, *Python*, …) sur micro-contrôleurs et la complexification des besoins en embarqué (IOT, exécution par des tiers, …) induisent de nouveaux besoins. (cf Stankovic 1988)

Langages fonctionnels

Les langages fonctionnels garantissent statiquement de bons invariants *logiques* de programmes, mais pas de bons invariants *de ressources*. Pourquoi?

Les fonctions d'ordre supérieur Le flux de contrôle du programme est dynamique, et donc plus dur à évaluer. En C ou équivalent, le graphe de flux de contrôle est genérable statiquement.

Structures de données Copy-on-write Même avec un algorithme simple type pipeline, on a pas d'assurance que l'implémentation va générer un programme travaillant en-place sur les structures déjà allouées.

Le ramasse-miette Il peut échouer à libérer la mémoire non-accessible dès qu'elle peut l'être, voire peut échouer à la libérer tout court, à cause de

cycles qui causent des fuites mémoires. Plus généralement, ils obscurcissent la gestion mémoire.

Sémantiques non-strictes Le coût d'un calcul ne dépend pas que de ça définition, mais aussi du contexte d'évaluation. Par exemple, la fonction

```
let null = fun [] -> true | _::_ -> false
```

n'a pas besoin d'évaluer entièrement son argument. Il est plus complexe de prendre en compte ces interaction appelant-appelé. Aussi, les thunks peuvent causer des fuites mémoires en maintenant en vie des structures le temps de leur non-évaluation.

Optimisations On veut garantir les consommation du programme à l'exécution à partir du code source. Entre les deux se tient un compilateur-optimisateur, qui à plus à faire pour un programme fonctionnel que qu'un compilateur C, par exemple Il faut donc adapter les passes d'optimisations pour obtenir et maintenir des propriétés utiles à la prévision statique de la gestion de ressources.

Contribution

On cherche à étendre l'application des langages fonctionnels à la programmation critique et embarquée en créant une analyse automatique des consommations en temps et mémoire des programmes fonctionnels. Il s'agit d'une analyse statique modulaire et dirigée par les types, dont le but est d'obtenir des bornes supérieures des la consommation mémoire. Ces bornes sont exprimés par des compositions de fonctions familières à croissance connues, prenant en arguments des grandeurs abstraites extraites des types de données du programme.

On introduit un langage sur lequel porte l'analyse, nommé *Hume*. Il est fonctionnel, à évaluation stricte, avec types et fonctions récursives, mais son cœur, *Hume Core*, est d'ordre un uniquement. Il est muni d'une machine abstraite à région mémoires et d'un ramasse-miette sans copie qui sert de modèle de coût. Cette machine est basée sur SECD et est étendue pour supporter des optimisations mémoires accommodantes pour l'analyse. Contrairement aux travaux précédents, on supporte les types et tailles définies par l'utilisateur.

On prendra note des faits suivants:

- L'analyse est restreinte à *Hume core*. Certaines constructions de Hume sont néanmoins admissibles dans ce cadre.
- On impose des contraintes linéaires entre les coût/les tailles. Cela permet de garantir une analyse automatique par des technique d'interprétation abstraite, comme l'accélération de point fixe dans le domaine abstrait des polyèdres.
- On ne supporte pas les calculs de coût en temps. Il sera simple d'ajouter un registre abstrait à la machine, incrémenté lors de l'exécution et de l'analyser. Mais l'hypothèse d'uniformité en coût d'une transition de la

machine ne tient pas nécessairement dans les systèmes réels. La machine abstraite de Hume garantie qu'une transition a toujours une consommation bornée, mais cette borne cache de nombreux détails d'implémentation. On pense par exemple aux lancements du ramasse-miette.

Exemple d'analyse

On considère un filtre simple, qui à un flux de flottant $x_0, x_1, ...$ renvoie un flux $y_0, y_1, ...$ défini par $y_i = \sum_{1 \leq k \leq n} w_k * x_{i-k}$. Dans une implémentation naïve en Haskell, il n'existe aucune garantie de libération des éléments du premier flux quand ils ne sont plus nécessaires. Le ramasse miette ne garantie pas non plus une borne précise de consommation mémoire pour cette fonction.

En Hume, on implémente un acteur synchrone qui génère le nouveau flux. La dépendance entre `pre xs` et `post xs` est rendu explicite par la clause `wire`. L'analyse statique détermine automatiquement que la liste qu'ils partagent est un invariant du programme.

```
type Float = float 32 -- 32-bit floating point numbers
box fir
 in (x::Float, xs::[Float])
 out (y::Float, xs'::[Float])
match
 (x, xs) -> (dotp [0.5,2,0.5] xs, x:init xs)
wire fir.xs fir.xs' initially [0,0,0]
```

En observant le type complet de `init` selon Hume,

$\forall n, m, s, h. \forall a. List^n a \xrightarrow{s,h} List^m a \ with :$
$n = m + 1, 0 \leq m, s \leq 6n - 3, h = 3n - 2$

on remarque:

- la présence de paramètres de tailles dans les types $List^k$ garantissant que l'argument est non-vide et que la sortie contient un élément en plus ($n = m + 1$).
- le type contient des informations de coût en pile et tas (s et h) et des bornes sur l'empreinte sur la pile et l'occupation mémoire. Ces propriétés ne sont pas purement dénotationnelles, mais demandent de fixer un modèle d'exécution: ici, c'est la machine virtuelle de Hume.
- Le type est polymorphique en ses taille, et indique donc que l'implémentation est de consommation uniforme. Le polymorphisme permet d'avoir une analyse modulaire. On doit faire attention: les optimisations (comme la fusion) peuvent faire mentir le type obtenu (en fusionnant les créateurs et les consommateurs de structure, on peut obtenir les tailles plus faibles).

On extrait de l'analyse de type un système d'inéquations linéaires qui est automatiquement calculé par approximation de point fixe sur un treilli, comme en interprétation abstraite. On obtient au final:

```
fir.xs : Size   [3,3], Stack   [1,19], Heap   [16,28]
```

Analyse de programme

Objectif: étendre le typage d'un langage pour y annoter des propriétés dynamiques des valeurs et des calculs. Raisonner ensuite dessus avec les outils de la théorie des types. On raisonne uniquement sur des structures définies inductivement, donc toujours modulo des équivalences.

Premier exemple

On considère l'extension de -calcul simplement typé avec exceptions, et un système d'effet modélisant ces exceptions: On a $\Gamma \vdash e : t\&\varphi$, avec φ un ensemble d'exception, défini par $\varphi = \varepsilon|\{e\}|\varphi \cup \varphi$. Les opérations ensemblistes y sont définies de manière usuelle. L'annotation des types des fonctions avec l'ensemble des exceptions qu'elle peut lancer induit une relation de sous-typage simple:

$a \xrightarrow{\varphi} b \prec c \xrightarrow{\psi} d$ iff $c \prec a \wedge b \prec d \wedge \varphi \subset \psi$

Et $\prec$ triviale autrement. Ce sous-typage est "shape-conformant", ce qui simplifie l'inférence de type. Mais une relation de sous-typage plus complexe peut améliorer la précision de l'analyse, en permettant de retarder l'affaiblissement des effets ($\varphi \subset \psi$) jusqu'au point d'appel de la fonction analysée.

On ajoute à notre système de type un `let` polymorphique, où l'on peut quantifier sur les effets et les types. C'est essentiel pour une analyse fine des paradigmes fonctionnels classique (plis, filtrage, mappage,...). On peut alors typer, par exemple, la composition de fonction:

`compose` : $\forall a, b, c.\forall \varphi, \psi.(a \xrightarrow{\varphi} b) -> (b \xrightarrow{\psi} c) -> (a \xrightarrow{\varphi \cup \psi} c)$

Pour notre système simple de lancement d'exception, le traitement du polymorphisme est insidieusement simple: En général, il faut restreindre les généralisation de type – c'est à dire la liaison des variables libres sous un pour les domaines des formes `let`. Sinon, l'inférence est trop complexe.

Inférence

L'inférence dans ce système d'effet exige de pouvoir "deviner" le type annoté d'expressions quelconque. Il nous faut donc un algorithme d'inférence de type et d'effet. On procède par *normalisation de preuves*: On restreint l'usage des règles de typage non-structurelles à certains noeuds syntaxiques où l'on sait inférer les types. Ici, on fait du *let-bound polymorphism*: les valeurs liées par des `let` sont

quantifiées autant que possible sur leurs types et effets à la déclaration, puis spécialisées au site d'utilisation.

Avec notre relation de sous-effets moins forte que le sous-typage, ou peut adapter l'algorithme W de Damas: on restreint les annotations d'effets sur les flèches à des variables fraiches uniquement, et on maintient séparément un ensemble de contraintes $\varphi \subset \psi$. Alors, l'unification des types devient un problème d'ordre un, pour lequel on peut utiliser W. Sans séparer les contraintes, c'est un problème d'unification modulo théorie des ensembles finis. Les contraintes d'inclusion des effets sont résolues par un solveur à part.

Étendre cette approche avec une relation de sous-typage exige "généralement" (sic.) d'ajouter des contraintes d'inégalité de types afin d'inférer un typage initial (au sens catégorique, c'est à dire "maximal"). La complétude de l'inférence de type dans ce cas reste un problème ouvert, mais des algorithmes aux résultats suffisamment corrects existent. Pour notre relation de sous-typage *shape-conformant*, un algorithme en deux temps permet d'obtenir un type pour tout terme, mais pas forcément son type initial. On procède en deux temps:

- Inférence des types (W) et collecte des contraintes $\varphi \subset \psi$,
- Résolution des contraintes et mise-à-jour des types.

Interprétation abstraite

C'est très classique, on mentionne vite fait les concepts déjà vu en TAS.

Il faut se souvenir des concepts classique en interprétation abstraite:

- Domaine abstrait en concret
- Structure de treillis sur ces domaines
- Connexion de Galois, et que faire quand il n'y en a pas
- Approximation de points fixes
- Opérateurs d'élargissement
- Treillis des intervalles
- Treillis des polyèdres convexes

La seule technique qui n'est pas explicitement au programme de *TAS* est *l'accélération à carburant*, qui permet de calculer des points fixes. On la présente maintenant. On commence l'itération de point fixe avec un nombre fini de carburant. Quand l'usage de l'élargissement causerais une perte de précision, on consomme un carburant et on fait une itération simple. Quand on a plus de carburant, on élargi même si on perd de la précision. Cela est utile pour la convergence des polyèdres. Voir *Bagnara et al., Generation of basic semi-algebraic invariants using convex polyhedra.*

Analyse statique de consommation de ressource

Analyse de complexité automatique

Préhistoire et système METRIC *METRIC* (1975) cherche a semi-automatiser les analyses manuelles de complexité temporelle asymptotique de programmes. Il suit de près les travaux séminaux de Knuth (1973). Il procède comme un humain:

1. Trouver une relation de récurrence sur le programme par transformation de source.
2. En extraire une récurrence sur la complexité, avec une métrique d'entrée pertinente.
3. Résoudre la relation, si possible.

Le résultat idéal est une formule fermée portant sur le coût des primitive, la longueur en tant que liste des arguments, et/ou leur taille en temps de S-expression.

Les métriques étudiés doivent être cumulatives et analytiques, donc on ne peut pas mesurer la taille maximale de la pile, qui a des mouvements de marée. On peut néanmoins compter le nombre de cellules `cons` allouées. Les types utilisateurs de sont analysés que comme S-expressions, et non selon leurs caractéristiques propres, comme la profondeur d'un arbre par exemple. Enfin, on notera que l'hypothèse d'accumulation du temps n'est valide que pour les langages a évaluation stricte. L'extension de l'analyse de ressource aux langages à sémantique non-stricte reste encore épineuse en 2020.

Le système ACE de Métayer (1988) Une nouvelle approche, basée sur la réécriture équationnelle. Le langage ciblé est fonctionnel, applicatif, à base de combinateur, et est donc bien adapté au problème. ACE produit une analyse au pire cas et asymptotique. Il procède par réécriture selon l'algèbre applicative et le *principe d'induction récursive* de McCarthy :

> Deux fonctions satisfaisant la même relatinon de récurrence sont égale sur le domaine défini par point fixe par le bas de la relation.
> — McCarthy

Le système de réécriture contient plus de 1000 règles entrées manuellement, sans étude (encore moins automatique) de leurs cohérence, ce qui aurait demandé bien trop de travail (on est en 1988 tout de même). L'analyse de dit pas si le terme asymptotique peut être dominé par les premiers termes, et ne supporte dans l'analyse de la consommation mémoire.

Rosendahl (1989) On utilise l'interprétation abstraite pour définir une transformation de programme permettant d'associer, pour un préfixe d'entrée, une borne supérieure de la consommation de temps.

Liu & Gomez (1998) Ils ont une approche semblable à du profilage: on peut

exécuter symboliquement le programme transformé pour obtenir des relations de récurrence sur le temps d'exécution à partir d'entrée partielle. On n'obtient pas de forme closes de ces relations. La délégation de la clôture des formules de récurrence est un leitmotiv du domaine. On fait le lien avec l'article d'Albert & al de 2008.

Walder (1988) Il brise le plafond de verre en proposant une analyse asymptomatique, et modulaire en présence de sémantique non-stricte, utilisant des *transformeurs de projection* décrivant la "paresse" des fonctions analysées. Il n'est pas allé jusqu'à un algorithme d'analyse.

Sands (1990) Il a créé plusieurs théories pour l'analyse des programmes fonctionnels avec fonctions d'ordre supérieur et évaluation paresseuse. Il peut ainsi obtenir des bornes [temps nécéssaire, temps suffisant] pour les fonctions paresseuses.

Ces formalismes ont pour but d'assister l'analyse asymptotique manuelle d'algorithmes, et donc ne considèrent ni l'automatisation totale de l'analyse et l'extension aux coûts opérationnels des programmes: on décompte toujours le nombre d'appel récursif d'une fonction

Système de type et d'effets pour le WCET

Dornic & al. (1992) proposent un "système de temps" pour un langage d'ordre supérieur à sémantique stricte. Il s'agit Une version spécialisé d'un système permettant de raisonner statiquement sur une classe de propriétés *intentionnelles* des programmes à l'exécution. Les jugements de typages sont annotés d'un coût arithmétique entier. Les flèches sont annotés d'un coût latent de leur exécution. Le système devient intéressant quand on peut quantifier sur les coûts latent, ce qui permet de typer le coût les fonctions d'ordre supérieur en fonction du coût latent de leur argument. Des travaux suivant ajoutent l'inférence des coûts.

Mais, ce système de temps est mis K.O. par la récursion, qui a un temps `long`, éléments absorbant du système. Aussi, l'absence de sous-typage des effets rend impossible la jointure de calculs de poids différents sans l'ajout manuel de poids morts pour équilibrer toutes les branches.

Ce système a été étendu par/Reistad & Gifford/ avec des annotation pour les tailles des entiers naturels, des listes et des vecteurs. Ces annotations sont des bornes statiques des tailles dynamiques des valeurs. Ils reconstruisent les types et effets dans un système d'effets algébrique. Par exemple:

$\texttt{succ} : \forall n. Nat^n \xrightarrow{1} Nat^{n+1}$
$\texttt{sub} : \forall n, m. Nat^n \times Nat^m \xrightarrow{1} Nat^n$
$\texttt{map} : \forall a, b, c, l. (a \xrightarrow{c} b) \times List^n(a) \overset{2+l*(3+c)}{\rightarrow} List^n(b)$
$\texttt{twice succ} : Nat^{\texttt{long}} \xrightarrow{7} Nat^{\texttt{long}}$

Le polymorphisme de taille sur les arguments, qui permet de donner un coût aux fonctions d'ordre supérieur usuelles. On doit surestimer les tailles des résultats des fonctions non-croissantes: on a Nat^n en résultat de `sub`, alors que Nat^{n-m}= est plus précis (et souhaitable). Enfin, le manque de polymorphisme cause des surestimations de tailles. Dans le dernier exemple, on ne peux par obtenir le typage souhaitable $Nat(n) \stackrel{7}{\rightarrow} Nat(n+2)$, par il faudra `succ` avec deux types différents.

Une solution, de Loild (1998), consiste à étendre le système avec les types intersections. Cela permet de spécialiser les fonctions polymorphiques sur plusieurs arguments en parallèle, et donc de typer la double application de `succ` de la forme $Nat^n \rightarrow Nat^{n+1} \rightarrow Nat^{n+2}$

Finalement, Vasconcelos & Hammond (2004) ont étendu cette technique aux définitions récurrentes, laissant le soin à l'utilisateur ou à un système d'algèbre automatisé de clore les relations de récurrences. Les problèmes de METRIC apparaissent aussi ici. On note aussi que l'approche se casse les dents sur les algorithmes diviser-pour-régner tels que quicksort.

Sized Types

Hughes, Pareto, Sabry (1996-2000)

Il existe des systèmes de types spécifiques à la déduction de propriétés de tailles, dédiés aux preuves de terminaisons ou aux optimisations. Hughes, Pareto, Sabry & al présentent en 1996 un système de type étendu aux tailles pour prouver la terminaison des programmes embarqués, et la propriétés *co*-rrespondante, la productivité des programmes co-récursifs (comme les streams).

Les types des constructeurs sont annotés avec des bornes supérieures des tailles des données construites pour les types récursifs, et des bornes inférieures pour les types co-récursifs. Ces bornes sont limités à l'arithmétique de Presburger pour des raison de décidabilité: donc pas de multiplication native.

```
// Les bornes sup sont indiquées entre ()...
zero : Nat(1)
succ :  i. Nat(i) -> Nat(i+1)

// ... et les bornes inf entre []
mk_stream :  i.  a. a -> Stream[i] a -> Stream[i+1] a
```

La relation d'ordre sur les tailles induit un sous-typage structurel sur les sized types. On note la taille arbitrairement grande. Donc, $\forall i. Nat(i) \subset Nat(\omega)$ On peut alors typer les expressions du genre

```
if cond then (??? : List(i) a) else (??? : List(j) a) : List(k) a
```

en sur-approximant la taille de la branche la plus petite. On fini avec $k = max(i, j)$ sans pour autant avoir l'opérateur max dans le système de type. La

régle de la récurrence (nouvelle) permet la récursion primitive sur les types à taille, garantissant la terminaison des fonctions récursives et la productivité des fonctions co-récursives.

On encode naturellement les récursions primitives sur les entiers et les listes, et on peut aménager un polymorphisme de taille pour gérer les accumulateur des fonctions tail-call récursives comme `reverse`. Mais il faut alors abandonner le polymorphisme classique pour garder la décidabilité de l'analyse.

Les schémas de récursion non-linéaire ne sont pas aussi bien gérés. Prenons l'exemple de `quicksort`, et de sa fonction auxilliaire `pivot`:

```
// type classique
pivot :  t. t -> list t -> list t * list t

// Meilleur type possible dans le système
pivot :  t.  i. t -> list(i) t -> list(i) t * list(i) t
quicksort :  t. list( ) t -> list( ) t

// On voudrait avoir
pivot :  t.  i,j. t -> list(i+j) t -> list(i) t * list(j) t
quicksort :  t.  i. list(i) t -> list(i) t
```

Il est hélas impossible de faire des inductions sur les sommes d'entiers dans ce système.

Hughes & Pareto (1999) : *Embedded ML*

Extension de leurs système à un langage de programmation à la ML, avec sémantique opérationelle à petit-pas basée sur la SECD, et une analyse des tailles des la pile et du tas. On ajoute une primitive d'allocation de mémoire `letregion #e in e` où `#e` est une nouvelle région mémoire de capacité dynamique, qui peut être remise à zéro. On y perd le coté *co*-récursif et donc l'analyse des streams, et ces efforts pour borner l'espace mémoire d'un instant de système synchrone, car la récursion peut créer des piles de régions.

Chin, Khoo (2001-2006)

Chin & Khoo ont créé un système avec inférence, basé sur l'arithmétique de Presburger, qui est décidable (en passant par Omega). Il s'intéresse à des propriétés de *sécurité*, pas de *liveliness* ou de *productivité* comme chez Hughes & Pareto. Il gère la récurrence avec une opération de *fermeture transitive des contraintes linéaires* permettant de "fermer la boucle": passer d'une étape de récurrence au calcul complet. Par exemple, le type inféré pour l'*append* classique est (les apostrophe désignent les arguments des appels récursifs):

```
append : list(m) t -> list(n) t -> list(l) t
avec:
  - m   0, n   0, l = m + n
```

```
  - m > m'   0, n' = n
```

Chaque type possède sa propre notion de taille: par exemple, les booléens sont de taille 0 ou 1. On peut alors typer `null` sur les listes de la manière suivante:

```
null : List(n) a -> bool(c)
avec:
  - (n = 0   c = 1)   (n > 0   c = 0)
```

Ca ressemble à ce qu'on peut faire avec des GADT il faudra regarder ce lien de plus près. On pourrait faire une équivalence entre les variables fantôme des GADT (empty | nonempty) et des invariants de tailles ($n = 0 \mid n > 0$). On ferra attention à ne pas oublier qu'on travaille à l'ordre supérieur.

La discipline de typage de Chin & Khoo n'a pas de preuve valide pour les types d'ordre supérieurs. Leur preuve implique l'existence de contraintes décrivant exactement la taille d'une valeur annoté d'un certain type. Ces contraintes n'existent pas à l'ordre supérieure.

Types Dépendants

Dependent ML

Créé en 1999, *dependent ML* (DML) est une extension d'OCaml avec types dépendants, relativement conservative. On y maintient la décidabilité du typage et on s'y efforce de maintenir les annotations dépendantes au minimum: DML sépare les valeurs classique d'OCaml des *indices* présents dans les types, qui sont pris dans un domaine de contraintes décidables. On peut par exemple utiliser des indices dans avec l'arithmétique de Presburger, qui sont ensuite résolus par Omega. L'évaluation des indices est donc limitée à la normalisation des contraintes.

Les types dépendants sont introduits par des raffinement de types afin de ne pas avoir à changer le code OCaml. `{v:T} U` introduit le produit $\Pi(v : T)U$ et `[v:T | P]` introduit la somme $\Sigma(v : T)P$. Grobauer a ensuite utilisé DML pour inférer des relations de récurrences sur les coûts des calculs, encore à résoudre ou faire résoudre par un solveur tiers.

```
append <| {m:nat}{n:nat} 'a list(m) * 'a list(n) -> 'a list(m+n)
filter <| ('a -> bool) * {n:nat} 'a list(n) -> [m:nat| m<=n] 'a list(m)
```

Avantage On peut définir nos propres notions de tailles, alors qu'avec les *Sized Types* la notion est rigide.

Inconvénient On ne résout par le problème de l'inférence, au contraire ! On est passé au problème de l'inférence des types dépendants…

Crary & Weirich: *LXres*

LXres est un langage de programmation avec type dépendants et code-comme-preuve permettant d'exposer des "horloges virtuelles" au niveau des types, et

donc de mesurer des coûts représentes par la progression de ces horloges. Ces estimations de coûts sont des *fonctions primitives récursives*, un formalisme puissant pour le problème en question, en comparaison, par exemple, à l'arithmétique de Presburger.

Brady & Hammond: typage à la *Epigram*

Suivant l'exemple de l'assistant de preuve *Epigram*, Brady & Hammond créés un langage dépendant avec un type `Size`. Les valeurs `size v p : Size A P` annotent les valeurs `v` indicés par une taille entière `n` et de type `A n`. La preuve `p : P` témoigne alors d'une propriété de taille sur `v`. On étend cette technique aux fonctions d'ordre supérieur en leurs associant des fonctions générant des `Size` pour des arguments d'ordre supérieurs. Mais ces approximations de taille pour les valeurs d'ordre supérieur ne sont pas inférées, il faut les obtenir manuellement.

Aussi, le système infère des propriétés de taille, donc dénotationnelles, mais ne considère pas l'obtention de propriétés intentionnelles, comme la taille du tas. On ne peut pas directement utiliser ces travaux pour obtenir des informations sur l'évaluation.

Cost Monad (Danielson)

Le plus simplement du monde:

```
return :: a -> Thunk 0 a
bind :: Thunk n a -> (a -> Thunk m b) -> Thunk (n+m) b
tick :: Thunk n a -> Thunk (S n) a
```

Des monades avec un indice dépendant pour la taille, implémenté dans Agda. On peut raisonner sur l'évaluation paresseuse en incluant directement des `Thunk` dans les structures de données. Il faut par contre bien connaître Agda pour obtenir des résultats : On est loin de l'analyse automatique de ressource. On peut néanmoins étendre cette approche aux coût dans une machine virtuelle. Encore une fois, pas d'inférence: on part dans la direction inverse.

Analyse amortie

On passe tout le blabla usuel sur Tarjan. Merci à lui quand même. On note l'existence de "Purely Functionnal Data Structures" de Okasaki, inspiré de sa thèse de doctorat de 1996.

Hoffman & Jost ont présenté dans *Type-Based Amortised Heap-Space Analysis* une méthode d'analyse amortie d'estimation de l'usage du tas par des programmes fonctionnels, dirigée par les types. On est encore dans le contexte d'un langage fonctionnel du premier ordre, avec évaluation stricte. Le langage libère explicitement la mémoire alloué dynamiquement par une indication syntaxique

sur les clauses `match`. Ce n'est pas très restrictif en pratique: GHC, par exemple, n'alloue que sur les `let` et ne libère que sur les `match`.

Dans cette analyse, les jugements et environnements de typage sont étendus par des coefficients de poids dans le tas, et des potentiels. On n'infère pas des tailles, mais les parts de la consommation du tas des différentes structures. Les estimations de taille mémoire ne sont obtenus qu'avec les tailles dynamiques (et donc inconnues) des structures en entrée et en sortie.

Exemple Le jugement suivant:

> `x : List(List(Bool, 1), 2), 3  e : List(Bool, 4), 5`
>
> signifie que `x` est une liste de liste de booléens, et que si `x` contient n éléments de tailles $t_1, ...t_n$, alors un tas de $3 + 2n + \sum t_i$ suffit à évaluer `e`, dont les m éléments occuperont $4m + 5$ cases du tas.

Cette analyse peut inférer les poids des types non-annotés. On peut donc la qualifier d'automatique. Pour ce faire, elle associe au programme un système d'équations linéaires dont les solutions sont les poids à inférer. On peut alors utiliser un système tierce de résolution linéaire pour obtenir les annotations.

Des limites: Les bornes de la taille du tas sont des expressions linéaires en la tailles des entrées-sorties. Mais on peut quand même diviser pour régner en découpant le potentiel. On manque de polymorphisme pour l'analyse des fonctions, qui prennent des poids fixes qui doivent correspondre simultanément à ceux de tout leurs site d'appels.

Enfin, étendre l'analyse à la pile est non-trivial. Les bornes en les tailles des structures sont linéaire, alors que l'occupation de la pile est souvent sub-linéaire. Campbell, dans sa thèse de 2008, étend ce système à la prise en compte de la *profondeur* des structures considérées.