Notes de lecture: *The Geometry of types*

Hector Suzanne

https://dl.acm.org/doi/10.1145/2480359.2429090

Introduction

On cherche à analyser la compléxité abstraite d'un programme, c'est à dire sans considérer finement le support matériel complexe sur lequel il est calculé. On peut ainsi obtenir des résultats *portables* sur une grande classe de cas d'utilisations des programmes analysés. Des travaux récents portent les systèmes de types linéaires.

Nous montrons que, pour un terme PCF t, la dérivation du typage $\vdash_I t : \sigma$ se réduit à la prouvabilité de d'un ensemble de formules arithmétiques du premier ordre, décrivant une *machine à jeton* pour t. La preuve ou infirmation de ces formules est un problème plus étudié, que des algorithmes et plateformes logicielles efficaces peuvent résoudre efficacement en pratique. Les bornes obtenues valent aussi pour le nombre d'étape de la réduction de t dans des machines abstraites modélisant la réduction CBV ou encore CBN du λ-calcul.

Nous obtenons alors une méthode d'analyse des programmes fonctionnels semblables à la logique de Hoare: on a la complétude et on peut générer des préconditions initiales (au sens catégorique). Mais à la différence de la logique de Hoare sur les programmes impératifs, notre analyse se compose naturellement car dirigée par les types (et non l'enchaînement causal).

La dépendance linéaire

La dépendance linéaire consiste à rendre compte dans les types du fait que différentes instanciations des valeurs typées peuvent recevoir des types différents. Par exemple, dans le jugement $f : (\mathbb{N}[a] \multimap \mathbb{N}[I]) \multimap^{c<2} (\mathbb{N}[a] \multimap \mathbb{N}[J])$, on reconnaît que f prend un argument fonctionnel envoyant un entier de taille a sur un entier de taille J, et en utilisant cet argument $c < 2$ fois, renvoie une fonction qui envoie un entier de taille a sur un de taille J. L'intérêt de la méthode est que a et c sont liés dans les formules I et J, et mieux, les différentes copies de l'argument (au plus 2) auront à leurs site d'appels respectifs les types $(\mathbb{N}[a] \multimap \mathbb{N}[I[1/c]]$ et $\mathbb{N}[a] \multimap \mathbb{N}[I[0/c]]$.

Contrairement au *sized-types*, on n'a pas à quantifier sur les index de tailles, mais l'on perd une forme de composition utile: dans notre approche, on doit typer les sous-termes dans leurs contexte. On fait de la *focalisation*, c'est courant en logique linéaire. Notre contribution est d'apporter une analyse composable d'inférence de type dans ce système.

Typer les termes dans notre système impose de trouver des bornes supérieures au nombre de copies qui peuvent en être faites, leurs *potentiels* respectifs. Nous faisons cela en considérant des modèles abstrait d'évaluation: la KAM de Krivine et la CEK et Friedman & Felleisen. Ces machines abstraites sont décomposables en une partie *sans duplication*, notamment la pile, et une *avec duplication*, notamment l'environnement. En étudiant les machines, le nombre d'étapes instanciant un terme est borné par son potentiel.

Typage

Indices

Notre langage des types est formé d'une couche d'annotation sur ceux de PCF. Les annotations sont des indices de potentiels, dont la grammaire est:

- $I = a, b, c, ...$ les variables d'indices
- $I = 0|1|I + J|I - J|...$ les opérations élémentaires sur
- $I = \sum_{a<I} K$ la somme bornée pour $a = 0, 1, ..., I$ des K
- $I = \odot_a^{I,J} K$ est la *cardinalité de la forêt K*, définie ci-dessous.

La cardinalité d'une forêt, notée $\odot_a^{I,J} K$ est le nombre de nœuds d'une forêt définie par les indices I, J, K: on considère que les nœuds des arbres sont étiquetés topologiquement, et que $K[n/a]$ est le nombre d'enfant du n-ième noeud selon cet étiquetage. On peut alors considérer le nombre de noeuds de la foret composée des J arbres dont la racine du premier est I de manière non ambiguë. C'est la *cardinalité de la forêt K en (I,J).*

Les cardinalités de foret permettent de décrire le potentiel des fonctions définies comme points fixes. La foret K représente les *unrolling* d'une fonction récursive à l'échelle globale, et l'opérateur défini plus haut permet de se focaliser sur la situation locale. On note que certaines cardinalité de forets sont indéfinies ou encore divergente, et correspondent aux points-fixes mal-formés ou divergents. Par exemple, le programme `let rec f x = f x in f 0` induit la cardinalité de foret $\odot_a^{0,1} 1$ pour l'appel final. La pseudo-foret correspondante contient un cycle qui mimique directement le cycle dans l'évaluation de `f 0`.

Donnés pour hypothèses un ensemble d'inégalitées sur les indices Φ, et un ensemble de variables libres dans les indices φ, et le programme $\mathcal{E}$ sur lequel porte l'analyse on définit le jugement $\varphi, \Phi \vDash_{\mathcal{E}} I \leq J$ pour signifier que I et J sont bien-formés et que les hypothèses dans Φ et la structure de $\mathcal{E}$ impliquent $I < J$ pour tout assignement des variables libres de φ.

Types

Notre systèmes différencie les types linéaires $A, B, ...$ (sans duplication) et les types *modaux* $\tau, \sigma, ...$ (avec dpulication). L'unique type linéaire est $\tau \multimap \sigma$ et les types modaux sont $\mathbb{N}[I, J]$ (le type des naturels de l'intervalle [I,J] abrégé en [I] quand I = J) et la *modalité* $[a < I].A$, qui équivaut au produit linéaire de I types $A_I \otimes A_{I-1} ... \otimes A_0$ où la variable a est instanciée dans A. Le type $[a < I].\tau \multimap \sigma$ est noté plus simplement $\tau \multimap^{a<I} \sigma$.

Afin de manipuler les types modaux algébriquement, on définit une projection sur les modalité: $[a < I + J].A = [a < I].A \cup [b < J].A[I + b/a]$. On sépare le produit des A en deux de ces composante, une à gauche avec les modalités "basse" et l'autre à droite les modalités "hautes". On étend ce concept aux sommes indicées définies (mollement) par $\sum_{a<I}[b < J].A = [a < \sum_{a<I} J].A$. On divise (ou fusionne) des morceaux de produits linéaires de A indicés.

Muni du jugement d'ordre sur les indices, on défini un sous-typage dans notre système: Il est simplement

- l'inclusion des intervalles dans
- la variance/contravariance pour les fonctions linéaires
- définit que $[a < I].A \leq [a < J].B$ si $J \leq I$ (contravariance) et $A \leq B$ sous hypothèse que $a < J$ (variance).

Jugement

Le jugement principal est $\varphi, \Phi, \Gamma \vdash^{\mathcal{E}}_{K} t : \tau$; avec

- φ, Φ comme plus haut les environnement des indices
- Γ un multi-ensemble linéaire donnant des types **modaux** aux identifieurs
- $\mathcal{E}$ est le programme équationel du programme entier
- K est un *indicde de poids*
- $t : \tau$ est le jugement final

L'environnement des termes Γ à pour support au moins les variables libres de t, et le partage des environnements et défini en étendant les unions/sommes sur les multi-ensembles en conservant la linéarité. Le système de type que nous donnons assigne un poids via la règle d'abstractions aux fermetures, et se poids est utilisé à la règle App.

Les seules parties non-inversibles du système de type sont les règles de sous-typage est les jugement sémantiques sur les indices utilisés en hypothèses de certaines règles. En réduisant ces parties à l'équivalence entre type et entre indices, on obtient un système donnant des jugement *initiaux* des programmes, depuis lesquels toutes les estimations moins précises des programmes peuvent être obtenues.

Le passage de CBV à CBN se fait en passant le type des entiers (en général tout les types primitifs) dans les types linéaires. On leurs impose alors une modalité

quand ils sont passé dans Γ, et c'est la règle App qui contient toute la mécanique de dotation/extraction de potentiel.

Enfin, on note que si l'on remplace les modalités "quantifiés" de notre systèmes par la modalité exponentielle ! de Girard et que l'on supprime les indices, notre PCF linéaire dépendant devient la traduction du λ-calcul dans la logique linéaire.

Machines abstraites

On modélise l'évaluation CBV (resp. CBN) par la machine asbtraite CEF (resp. KAM). On peut alors donner précisément un coût d'évaluation aux programmes par la sémantique opérationelle à grand pas $t \Downarrow^n_? u$ avec $? =$ CBV (resp. $? =$ CBN). Les théorèmes suivants montrent l'adaptation de notre système à la mesure de coût d'exécution des programmes.

Bornage Si $t \Downarrow^n_{\text{CBV}} u$ et $\vdash^{\mathcal{E}}_K \ t : A$ alors la sémantique de I dans $\mathcal{E}$ est une borne sup. de n. Idem dans CBN avec $t : [a < 1].A$

Conservation $\varphi, \Phi, \emptyset \ \vdash^{\mathcal{E}}_I \ t : \tau$ et $t \to u$, alors $\varphi, \Phi, \emptyset \ \vdash^{\mathcal{E}}_J \ t : \tau$ pour un certain J et $\varphi, \Phi \vDash I \leq J$.

Correction $\varphi, \Phi, \emptyset \ \vdash^{\mathcal{E}}_I \ t : \mathbb{N}[I, J]$ alors, en notant i, j, k les sémantiques des indices sous $\mathcal{E}$, on a $i \leq n \leq j$ et l'évaluation de t prend au plus $|t| * (k+1)$ étapes.

Complétude Avec encore k, m la sémantique de K, M sous $\mathcal{E}$, si $t \Downarrow^k_{\text{CBV}} m : \mathbb{N}[M]$, alors $\vdash^{\mathcal{E}}_K \ t : \mathbb{N}[M]$ est dérivable. Le résultat tient encore pour les fonctions de dans , et donne des indices avec une variable libre bornant la sortie et son coût en fonction de l'entrée.

Ces résultats forts sont à nuancer par deux points:

- La sémantique des indices se fait modulo le programme équationel $\mathcal{E}$, qui doit permettre de représenter toutes les fonctions récursives partielles (FRP) par un indice.
- Tout les jugements sémantiques $\varphi, \Phi \vDash_{\mathcal{E}} I \leq J$ corrects sont prouvables.

Ces deux faits, pris ensembles, implique que notre système doit pouvoir décider le bornage asymptotique d'une FRP par une autre, ce qui bien entendu impossible, car on se heurte alors frontalement au théorème d'imcomplétude.

Mais, pour un programme PCF donné, nous n'avons pas besoin de tout ce pouvoir expressif. Dans la pratique, on peut obtenir à programme donné un programme équationel $\mathcal{E}$ suffisamment puissant pour sa dérivation sans être universel. On peux alors collecter les jugements sémantiques pour les offrir à un oracle, qui, s'il est concluant, permet d'obtenir la dérivation.

Inférence modulo oracle

Schéma général

L'algorithme d'inférence est plus précisément un algorithme inférant des contraintes et un programmes équationnel, qui doivent en suite être résolues pour conclure. On procède en trois temps:

1. Inférer le type principal d'un terme PCF t (classique)
2. En travaillant sur la dérivation du type, raffiner les types en types linéaire, produire le programme $\mathcal{E}$ et un ensemble de contraintes $\mathcal{J}$
3. Vérifier la prouvabilité de $\mathcal{J}$ modulo $\mathcal{E}$

Le jugement racine de la dérivation linéaire est $a; \emptyset; \emptyset \ \vdash_I^{\mathcal{E}} \ \mathbb{N}[a] \multimap^1 \mathbb{N}[J]$.

Il est alors possible, pour une fonction de coût quelconque, de savoir si elle borne bien le coût de t en ajoutant la contrainte à $\mathcal{J}$, et ce sans avoir à recalculer parties 1-2 de l'algorithme. Pour montrer que le WCET de tn est borné par $f(n)$, il suffit de rajouter $I \leq f(a)$. Ce schéma s'étend aux fonctions d'ordre supérieur et à l'évaluation CBV, etc… On a bien réduit le calcul de complexité à un problème de résolution de formules entières du premier ordre équivalent.

Préliminaires

On peut fortement **limiter le sous-typage** en imposant des équivalences entre les environnements/types/indices. La règle de sous-typage produit uniquement des contraintes de la forme $\varphi, \Phi \ \vDash^{\mathcal{E}} H_i \leq K_i$ avec les H_i et K_i présents dans l'instance de la règle de sous-typage, et sont équivalentes à une *condition de bord* imposant que H_i représente une fonction récursive totale sur ses variables libres. Ces conditions de bords sont accumulées en étant supposées valides durant l'inférence, et sont vérifiées en bloc par la suite.

L'algorithme limite parfois l'usage des indices à ceux formant des **types primitifs**, c'est à dire des types ne contenant que des symboles de fonctions du langage des indices, saturés par des variables libres. Les symboles de fonctions reçoivent une sémantique via le programme $\mathcal{E}$, qui est un système de réécriture composé d'un ensemble de définitions $f(a_1, ..., a_n) = J$, où les variables libre de l'indice J sont toutes des a_i. Un symbole de fonction peut être spécifié ou pas par un programme $\mathcal{E}$ durant la dérivation, et notre algorithme fonctionne justement par ajout successif de définition au programme équationnel. Un jugement dans notre système de type est bien formé si tout les symboles de **polarité +** sont définis en fonction de ceux en **polarité -**.

Enfin, nous implémentons des axiomes classiques de la **logique linéaire** dans notre système, afin de pouvoir donner un sens aux principes comme la déréliction ou la contraction avec nos structures de données. Les axiomes classiques que nous transposons par ce biais dans notre système quantifiés sont:

Déréliction En logique linéaire, $!A \to A$. Pour nous, la déréliction produit

pour une instance focalisé d'une paire de type $([a < 1].A, A[0/A])$ les conditions de bord et programme équationnel partiel encodant leur équivalence.

Contraction Classiquement, $!A \to !A \otimes !A$. Notre algorithme de contraction permet de traduire la séparation d'une union disjointe de modalité dans un couple conditions de bord/programme équationnel.

Déroulement Idem, mais pour les sommes de modalités indicées par des indices. Cela traduit $!A \to !!A$.

Affaiblissement La destruction par affaiblissement des types linéaire se traduit par la création d'un programme équationnel partiel définissant sur les bonnes polarités les symboles encore libre des types effacés. C'est une procédure simple: comme les types focalisés sont effacés, on peut définir ces symboles arbitrairement. C'est néanmoins important que ces symboles soient **définissables** sur les polarités demandées.

Algorithme d'inférence

L'algorithme d'inférence a pour signature:

$$\texttt{GEN}(\varphi, \Phi, d_{PCF}) = (\Gamma \vdash_I t : \tau, \mathcal{E}, \mathcal{R})$$

,

avec d_{PCF} la dérivation du type de $t$dans PCF non-linéaire, $\mathcal{E}$ un programme eéquationnel *partial*, et $\mathcal{R}$ un ensemble de condition de bord. L'algorithme admet deux invariants importants:

Decoration La racine de la dérivation est la version non-linéaire du jugement en sortie

Polarité La polarité du jugement en sortie est correcte (i.e. les polarité des symboles de fonctions introduits sont bien celles des objets qu'ils représentent).

L'algorithme fonctionne en deux passes:

1. Dans un premier temps, il décore inductivement la racine de la dérivation pour la rendre linéaire,
2. Il effectue récursivement les deux phases sur les enfants de la racine,
3. Puis, en utilisant les algorithmes auxiliaires dérivés de la logique linéaire, il déduit des $\mathcal{E}$ et $\mathcal{R}$ produits sur les enfants le programme équationnel et les conditions de bord à la racine.

Pour cette dernière étape, on utilise sans trop rentrer dans les détails:

- L'affaiblissement pour signifier l'effacement des environnements aux feuilles de dérivation (constantes, variables)
- La contraction pour l'appel de fonction, pour séparer les coûts entre le coût de la fermeture, celui de l'argument, et enfin celui de l'appel

- Le déroulement pour la création de fermeture récursive, afin de séparer le coût abstrait de la fermeture dans les appels récursifs.

La cohérence est complétude relative de l'algorithme ont étés établies, et donc il es garantie que la dérivation linéaire est correcte et précise si et seulement si le programme équationnel final garantie la validité des conditions de bords (qui contiennent les fonctions qu'il définit).

Implémentation et perseptives

Nous avons réalisé une implémentation complète dans OCaml, et l'on note les faits suivant. On a uniquement besoin de créer des symboles de fonctions quand le (sous-)jugement considéré assigne un type d'arité non-nulle au (sous-)terme correspondant. De nombreuses contraintes de bords sont trivialement satisfiables ou simplifiables, au point ou une gestion ad-hoc de ces équation semble être une piste de recherche pertinente.

Le problème décrit par la sortie de l'algorithme est de compléxité identique à celui en entrée, donc indécidable dans le cas général, mais est décrit dans un langage d'ordre 1 uniquement. Dans le cas ou une résolution automatique n'est pas faisable, nous avons implémenté une traduction des paires $(\mathcal{E}, \mathcal{R})$ vers des théorie dans l'assistant de preuve *Why3*.

Les systèmes de types proposés auparavant pour l'analyse de compléxité du λ-calcul, par exemple ceux garantissant une compléxité polynomiale, sont très restrictifs: ne nombreux termes ayant cette compléxité sont rejetés. Notre propriété de complétude relative est nouvelle, à notre connaissance.

Les analyses statiques de coûts sont soient limités à certaines métriques de coûts, ou encore des bornes multi-linéaires, et parfois ne suppriment pas l'ordre supérieur.

Notre stratégie reprend l'objectif de la logique linéaire dépendante: de reformuler les problèmes de jeux et de stratégies en problèmes de typage. Notre approche générale pose des bases qui méritent d'être étudiés pour des applications à des cas particuliers, et des intégrations avec des solveurs moderne, automatiques ou interactifs. De nombreux aspects techniques restent à considérer.