Sur l'espace des termes et des machines

Sambo Boris Eng
Université Paris Diderot - Paris 7

Mémoire de M1 Informatique supervisé par Damiano Mazza

Résumé

Suite aux travaux de Beniamino Accattoli et d'Ugo Dal Lago rapprochant la complexité en temps du λ-calcul et celle des machines de Turing, nous nous adressons à la question de l'espace et de la possibilité de transposer et d'analyser ses résultats dans le paradigme fonctionnel. Dans cet article nous faisons usage d'outils provenant de la logique linéaire comme la géométrie de l'interaction pour proposer une simulation en espace constant des termes aux machines. Nous montrons aussi que similairement au fait qu'une machine induit une famille de circuits booléens que l'on peut générer efficacement, il en est de même pour les λ-termes. Ces résultats forment un premier pas vers la pertinence du λ-calcul dans le monde la complexité pouvant amener avec lui de nombreux outils de la programmation et de la logique afin de donner une nouvelle vue sur la complexité des programmes.

Prérequis : logique booléenne, notions élémentaires de calculabilité et de complexité

1 Introduction

1.1 Motivations

Qu'est-ce que le « calcul » ?

Cette question fondamentale qui a occupé les mathématiciens d'antan puis été délaissée aux informaticiens d'aujourd'hui a trouvé sa réponse dans diverses modélisations abstraites qu'on perpétue aujourd'hui encore. La première approximation du calcul apparaît en 1936 avec les *machines de Turing* [Tur37] prenant la forme d'une machine imaginaire écrivant et lisant des symboles séquentiellement sur les cases d'un ruban infini : on capture la partie mécanique et opérationnelle du mathématicien tout en s'abstrayant de tout le reste qui est inutiles afin de pouvoir raisonner aisément dessus. C'est la vision **mécanique** du calcul.

À partir de là ont émergé de nombreuses alternatives comme les *circuits booléens* [Vol99] et le λ-calcul [Chu32] proposant tous deux une vision très différente du calcul. Il se trouve que tous ces modèles sont en fait équivalents : c'est la célèbre « thèse de Church-Turing » entretenant aujourd'hui un statut plus philosophique que technique. C'est la vision **algébrique** du calcul.

Vu cette équivalence malgré cette grande disparité dans la définition du calcul, nous sommes donc en droit de nous demander : qu'est-ce qui les distingue ? Il semblerait que la réponse se trouve dans le domaine de la *complexité algorithmique*. Grâce à leur finesse opérationnelle et leur manipulation intuitive, les machines de Turing deviennent le modèle standard permettant de poser un premier cadre à la notion de complexité du calcul c'est-à-dire le *temps* et l'*espace* consommé afin d'aboutir à la solution d'un problème. C'est une première porte vers de nombreuses interrogations comme le célèbre problème $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ [Coo71] qu'on ne sait toujours pas aborder à l'heure actuelle et ce depuis des dizaines d'années.

De grands murs étant levés, il est utile de comprendre les corrélations entre les modèles de calcul. Chaque modèle bénéficiant de sa propre communauté scientifique, un rapprochement en complexité de deux modèles rapproche aussi des travaux, des outils et des personnes.

Dans les premiers pas de la complexité, a été suggéré l'existence d'une corrélation entre la complexité des modèles les plus courants : c'est la « thèse d'invariance » [vEB90].

1. D'une part, dans l'optique de la thèse d'invariance, nous faisons le choix de développer le rapprochement entre deux modèles connus : la machines de Turing (mécanique) et le λ-calcul (algébrique) à travers leur **complexité en espace**.
2. D'autre part, il est connu dans la littérature que les circuits booléens (vision statique du calcul) peuvent approximer les machines de Turing (vision dynamique du calcul) dans une sorte de dualité hardware/software. Il semblerait qu'une telle approximation a son analogue dans le λ-calcul. C'est aussi une chose que nous voulons investiguer.

1.2 Machines de Turing

Les machines de Turing sont des machines abstraites fonctionnant avec un *ruban infini* composé de cases, d'une *tête de lecture* pointant sur la case courante et de règles de transition dictant ce qu'il faut écrire et s'il faut se déplacer à gauche ou à droite sur le ruban.

Definition 1 (machines de Turing).

Une machine de Turing déterministe $\mathfrak{D}$ est un 7-uplet $(Q, \Gamma, b, \Sigma, q_0, F, \delta)$ où :

- Q est l'ensemble des états de la machine
- Γ sont les symboles traitables, $\Sigma \subseteq \Gamma$ sont ceux autorisés sur l'entrée et $b \in \Gamma$ est le symbole représentant le vide remplissant le ruban au départ
- $q_0 \in Q$ et $F \subseteq Q$ sont respectivement l'état initial et l'ensemble des états finaux
- $\delta : (Q \backslash F) \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$ est la *fonction de transition* décrivant les règles de transition

On peut représenter une machine par un graphe d'état accompagné d'un ruban.

La simplicité de ces machines en font un modèle standard pour la complexité : le temps est quantifié avec le nombre de transitions et l'espace avec le nombre de cases utilisées.

1.3 λ-calcul pur

Le λ-calcul noté Λ, proposé par Alonzo Church dans les années 1930 [Chu32], donne corps aux fonctions abstraites des mathématiques. La fonction $f(x) = y$ devient une entité syntaxique, explicite $\lambda x.y$: une banale séquence de caractères. Ce formalisme n'est plus la liaison entrée-sortie f mais la description de son fonctionnement en terme de connectiques primitives.

Definition 2 (calcul Λ).

Les *termes* (aussi appelés *programmes*) du λ-calcul émergent de la grammaire suivante :

$$M, N := x \mid \lambda x.M \mid MN$$

C'est à dire que l'on a des *variables* x, des *abstractions* (fonctions) $\lambda x.M$ et des *applications* (passage d'une entrée à une fonction) MN.

Definition 3 (réduction de Λ).

On peut faire un pas d'évaluation pour les programmes en faisant usage d'une procédure de réécriture syntaxique nommée β-réduction :

$$(\lambda x.M)N \longrightarrow_\beta M\{x := N\}$$

Elle remplace toutes les occurrences de x dans M par N : c'est ce que font implicitement les fonctions mathématiques. On omettra souvent le β pour écrire simplement $\longrightarrow$.

Definition 4 (évaluation de Λ).

L'évaluation complète d'un terme $M \in \Lambda$ est donnée par la clôture réflexive-transitive de $\longrightarrow$ notée $\longrightarrow^*$.

Ce nouveau paradigme donnera lieu plus tard à la *programmation fonctionnelle* où les programmes ne sont pas des séquences d'instructions mais des descriptions permettant de produire un résultat à partir d'une entrée.

1.4 Contributions et méthodes

Travaux passés. Précédemment, Beniamino Accattoli et Ugo Dal Lago ont montré qu'il y avait une manière d'évaluer les termes du λ-calcul (par *réduction de tête*) de telle sorte à ce que la complexité en temps du calcul soit proche, à un polynôme près, de celle des machines de Turing [AL12]. Du côté de la correspondance machine/circuit vue dans les termes, il n'y a apparemment aucun travail explicite à ce sujet. Nous sommes principalement inspirés par de vieux résultats sur les circuits comme ceux de Borodin [Bor77] que nous transportons dans le λ-calcul.

Nos contributions et méthodes. De façon duale, nous nous intéressons à une manière de mesurer l'espace du λ-calcul. Le λ-calcul ayant un mécanisme complexe au potentiel infini on adopte la méthode mathématique d'approximer les phénomènes infinis par des constructions maîtrisées (e.g algèbre linéaire). Nous commençons par décomposer la mécanique du λ-calcul grâce à la logique linéaire [Gir87], puis définissons une notion d'approximation par des termes statiques et finis sur lesquels on appliquera la procédure de géométrie de l'interaction [Gir89]) qui nous permettra de travailler avec un espace logarithmique sur les termes.

1. Notre contribution principale est un théorème de simulation en espace constant du λ-calcul dans les machines de Turing. Nous prenons comme point de départ le théorème 4 de [Mazb] que nous raffinons et complétons.
2. Nous complétons avec une preuve de l'uniformité des termes induits par la simulation. C'est-à-dire que tous comme les circuits, le λ-calcul induit une famille de termes pouvant être générée par un programme. C'est un résultat existant entre machines et circuits que nous transportons dans les termes.

Travaux connexes. Dans [SBHG08] on se focalise sur l'implémentation et une description de bas niveau pour quantifier l'espace des termes fonctionnels. D'un autre côté, Schöpp [Sch06] avait déjà proposé d'utiliser la géométrie de l'interaction pour évaluer les λ-termes efficacement en espace. C'est un point essentiel dont nos résultats se servent et que nous complétons avec les travaux de Mazza sur les approximations polyadiques [Maz17b].

2 Complexité et décidabilité

Commençons par comprendre comment les machines de Turing et le λ-calcul peuvent utiliser une notion calcul qui leur est propre afin de résoudre des problèmes. Ensuite, nous expliquons comment chaque modèle définit sa propre notion de complexité en déterminant un moyen de mesurer le calcul autant en temps qu'en espace.

2.1 Décidabilité des machines

En général on utilise un codage binaire par convention afin de coder l'entrée des machines (avec éventuellement des séparateurs ou symboles utilitaires).

On définit la résolution d'un problème par la détermination de l'appartenance d'un mot à un langage formel. Par exemple vérifier si le codage binaire de $< G, s, t >$ appartient à

$$\texttt{PATH} = \{< G, s, t > |\ \text{il y a un chemin de } s \text{ à } t \text{ dans le graphe } G\}$$

revient à résoudre un problème d'accessibilité de sommets dans un graphe.

Definition 5 (acceptation des machines).

> Une machine $\mathfrak{D} = (Q, \Gamma, b, \Sigma, q_0, F, \delta)$ lit une entrée $w \in \{0,1\}^*$ caractère par caractère et applique δ jusqu'à atteindre un état $q \in Q$ avec un résultat sur le ruban. Si $q \in F$ alors on dit que la machine *accepte* w (noté $\mathfrak{D}(w) = 1$) sinon, on dit qu'elle le *rejette* (noté $\mathfrak{D}(w) = 0$). La machine peut aussi éventuellement tourner à l'infini.

Definition 6 (décision des machines).

> Une machine $\mathfrak{D}$ *décide* un langage $\mathcal{L}$ quand
>
> — si $w \in \mathcal{L}$ alors $\mathfrak{D}(w) = 1$
>
> — si $w \notin \mathcal{L}$ alors $\mathfrak{D}(w) = 0$
>
> La machine $\mathfrak{D}$ est ainsi appelée décideur de $\mathcal{L}$. S'il existe un décideur pour un langage $\mathcal{L}$, on dit qu'il est décidable. En particulier, la décidabilité ne concerne que les machines qui s'arrêtent.

2.2 Complexité des machines

Definition 7 (complexité en temps des machines).

> La complexité en temps d'une machine $\mathfrak{D}$ est donnée par une fonction $t : \mathbb{N} \longrightarrow \mathbb{N}$ telle que sur une entrée de taille n, le calcul de $\mathfrak{D}(w)$ effectue au plus $t(n)$ étapes de transition.

Definition 8 (complexité en espace des machines).

> La complexité en espace d'une machine $\mathfrak{D}$ est donnée par une fonction $s : \mathbb{N} \longrightarrow \mathbb{N}$ telle que sur une entrée de taille n, le calcul de $\mathfrak{D}(w)$ utilise au plus $s(n)$ cases du ruban.

Definition 9 (notations de Landau).

> Comme la complexité exacte des machines est trop complexe, on se contente, en général de l'approximer par une borne asymptotique en utilisant les *notations de Landau*. On utilisera seulement la notation $f = O(g)$ pour dire que g domine asymptotiquement f, c'est-à-dire que $g(n)$ est une borne ultime de $f(n)$ quand on s'étend vers des valeurs de plus en plus grandes pour n.

Quand on dit qu'une machine tourne en temps $O(f)$ cela signifie que son temps d'exécution est borné par $f(n)$ pour une entrée de taille n. En particulier on s'intéresse souvent aux bornes logarithmiques $O(log\ n)$, linéaires $O(n)$, polynomiales $O(n^k)$ et exponentielles $O(k^n)$.

Comme on travaillera souvent avec l'espace logarithmique, on donne une définition de la calculabilité d'une fonction en espace logarithmique.

Definition 10 (log-constructibilité en espace).

> Une fonction $f : \mathbb{N} \to \mathbb{N}$ est log-constructible en espace si et seulement s'il existe une machine de Turing M telle que sur l'entrée 1^n, elle termine sur $f(n)$ en sortie avec :
>
> — une utilisation du ruban bornée par $log(f(n) + 1)$ en espace
>
> — $f(n)$ est borné par $log(f(n) + 1)$

2.3 Décidabilité des termes

Le cas des termes se complique un peu. Similairement aux machines de Turing, on doit trouver un codage des chaînes binaires mais aussi des booléens dans le langage du λ-calcul. Une fois que cela est fait, la décidabilité devient plutôt naturelle.

Definition 11 (codage des booléens).

> Nous utilisons le codage de Church qui existe depuis le début du λ-calcul.
>
> $$\underline{1} = \lambda x.\lambda y.x \qquad\qquad \underline{0} = \lambda x.\lambda y.y$$
>
> Cela correspond à la première et seconde projection des arguments dans Λ.

Definition 12 (codage des chaînes binaires).

> Il existe plusieurs codages des chaînes binaires mais nous faisons le choix d'utiliser celui proposé par Mogensen [Mog01]
>
> $$\underline{w_1 ... w_n} \in \{0,1\}^n = \lambda s_0.\lambda s_1.\lambda e.\underbrace{s_{w_1}(...(s_{w_n}\ e)...)}_{\text{n}}$$
>
> Par exemple on a :
>
> $$\underline{0} = \lambda s_0.\lambda s_1.\lambda e.e \qquad \underline{1} = \lambda s_0.\lambda s_1.\lambda e.(s_1\ e) \qquad \underline{10} = \lambda s_0.\lambda s_1.\lambda e.s_1(s_0\ e)$$

Definition 13 (acceptation des termes).

> Un terme $M \in \Lambda$:
>
> — accepte un mot $\underline{w}$ si $M\ \underline{w} \longrightarrow^* \underline{1}$
>
> — refuse un mot $\underline{w}$ si $M\ \underline{w} \longrightarrow^* \underline{0}$
>
> — peut aussi boucler à l'infini dans sa chaîne de réduction

Definition 14 (décision des termes).

> Un terme $M \in \Lambda$ décide un langage $\mathcal{L}$ quand
>
> — si $w \in \mathcal{L}$ alors M accepte $\underline{w}$
>
> — si $w \notin \mathcal{L}$ alors M rejette $\underline{w}$
>
> Une fois de plus, on ne considère ici que les termes qui terminent (comment les caractériser ?).

2.4 Complexité des termes

Pour le cas des termes, les choses deviennent bien moins évidentes. On peut tout de même tenter une première approche naïve.

— Pour le temps, le premier réflexe est de prendre le nombre de β-réductions. Cela marche, mais est-ce que c'est satisfaisant ?

— Pour l'espace, c'est encore plus flou. La chose la plus naturelle est de choisir la taille du plus grand terme apparaissant au cours de la réduction. Mais l'espace étant au moins la taille du terme de départ, où est l'espace logarithmique ?

Comme énoncé dans les écrits de Beniamino Accattoli, cette façon de mesurer le temps et l'espace sont problématiques.

2.5 Modèles de coût

Definition 15 (modèle de coût).

> Un modèle de coût est une façon de mesurer le temps et l'espace dans un modèle de calcul. Par exemple, dans la section précédente, nous avons défini un modèle de coût pour le λ-calcul.

Il s'agit maintenant de comprendre si le λ-calcul constitue ce que l'on appelle un *modèle de coût raisonnable*, c'est-à-dire un modèle de calcul proche, en terme de complexité, du modèle standard, les machines de Turing.

Definition 16 (modèle de coût raisonnable).

> Un modèle de coût est raisonnable si et seulement si, par rapport au modèle de coût standard des machines de Turing, il a
>
> — un écart **polynomial** en **temps**
>
> — un écart **constant** en **espace**
>
> On se laisse un peu de liberté pour le temps à cause du fait que le temps polynomial représente le calcul *faisable, réaliste en pratique*. Dans le cas contraire, sachant qu'une machine à deux rubans produit un écart de temps au plus polynomial, il serait absurde de dire qu'elle n'est pas raisonnable.

Une implication intéressante de cette définition est qu'un modèle de coût raisonnable permet de mixer les résultats de deux modèles de calcul. Par exemple, Mazza a pu prouver le théorème de Cook-Levin dans le cadre du λ-calcul [Maz16].

Voyons maintenant pourquoi le modèle de coût donné précédemment pour le λ-calcul n'est pas raisonnable et comment y remédier :

— Pour le temps, si l'on prend un terme $D \coloneqq \lambda d.\lambda a.dd(aa)$ et un terme A quelconque alors on a la réduction suivante :

$$DDA \longrightarrow^* DD(AA) \longrightarrow^* DD(AA(AA)) \longrightarrow^* DD(AA(AA)(AA(AA)))$$

Cette duplication exponentielle découle du fait que la β-réduction n'est pas une opération atomique. Un nombre linéaire d'étapes induirait ainsi un temps exponentiel. Ce n'est pas une bonne mesure. Beniamino Accattoli et Ugo Dal Lago se sont chargés du soucis en utilisant la réduction de tête comme modèle de coût raisonnable [AL12].

— Quant à l'espace, on a toujours une explosion exponentielle en taille. Le temps étant toujours supérieur ou égal à l'espace (pour traverser une unité d'espace, il faut une unité de temps), cela induit une exponentiation du temps. Contradiction. Nous abordons une solution dans cet article.

Pour avancer dans la question de l'espace il faut passer au cadre dit « typé » du λ-calcul qui va nous permettre de :

— caractériser la classe des termes qui terminent en les contraignant avec un type. On éliminera ainsi les termes qui ne se terminent pas par absence de typage possible.

— mesurer de façon plus fine la complexité des termes avec des informations quantitatives utiles contenues dans le type.

Nous proposons de voyager dans le monde de la logique où types et formules ne font qu'un en plus de suggérer une analyse fine des propriétés quantitatives du calcul. Notre but sera de décomposer la mécanique du λ-calcul pour la rendre plus facile à manipuler puis de définir une notion *d'approximation des termes* afin de pouvoir utiliser les outils qui nous intéressent.

3 Raffinement du calcul

Pour étudier l'espace du λ-calcul pur Λ, nous donnons un système équivalent mais avec une mécanique plus fine et précise en plongeant à travers la *logique linéaire*.

Cependant, il est à savoir que ce passage n'est pas nécessaire, et ne fait que simplifier l'extraction de constructions finies pour étudier la consommation en espace du calcul.

3.1 Isomorphisme de Curry-Howard

L'isomorphisme de Curry-Howard, rendue explicite dans les années 80 mais déjà ressentie bien avant [How80], établit une correspondance formelle entre logique et calcul par les équivalences suivantes où $M : A$ indique que le terme M est de type A :

Logique	**Calcul**
Formule A	Type A
Preuve de A	Programme $M : A$
Simplification des preuves	Évaluation des programmes

La correspondance est directement visible avec la célèbre règle du Modus Ponens établie depuis le temps d'Aristote et de la philosophie stoïque :

$$\frac{A \Rightarrow B \quad A}{B}\ (MD) \qquad \frac{M : A \Rightarrow B \quad N : A}{MN : B}\ (\lambda_{app})$$

Cette correspondance démontre la pertinence de l'usage de la logique dans le calcul. Les règles logiques forcent un format normatif, restrictif, contraignant le comportement des termes à une certaine cohérence subjective. Par exemple, plus tard nous introduirons un système de type (et donc une logique) qui capture la classe des termes qui terminent. Mais avant cela continuons d'explorer les entrailles du calcul.

3.2 Logique linéaire

La logique linéaire découverte par Jean-Yves Girard [Gir87] a un statut hybride puisqu'elle provient du monde de la logique sans vraiment y faire sens.

On peut voir ce système logique comme une décomposition de la logique usuelle où l'on a séparé la formule de son caractère statique et infini qui devient maintenant explicite. Par caractère infini on entend les propriétés dîtes « structurelles » suivantes :

$$\frac{A \text{ prouvable} \quad A \text{ prouvable}}{A \text{ prouvable}} \qquad \frac{A \text{ prouvable}}{A, B \text{ prouvable}}$$

Lisons les règles de bas (conclusion) en haut (hypothèses). La première, appelée *duplication* est responsable du caractère éternel de la vérité et la seconde, appelée *effacement*, est responsable de l'émergence de vérités qui ne sont pas directement causales (par exemple le faux implique le vrai, tout implique les évidences).

La logique linéaire sépare les formules en deux classes :

— les formules linéaires A qu'on ne peut ni dupliquer, ni effacer.

— les formules exponentielles $!A$ qui sont les formules usuelles.

Cette séparation n'est pas sans conséquence puisqu'elle décompose la logique qu'on a toujours eu l'habitude d'utiliser :

Logique habituelle	A	$A \wedge B$	$A \vee B$	$A \Rightarrow B$
Logique linéaire	$!A$	$A \otimes B \quad A \& B$	$A \parr B \quad A \oplus B$	$!A \multimap B$

On remarque que $!A \multimap B$ explicite le fait qu'une fonction peut utiliser son argument un nombre illimité de fois. C'est en fait de ce constat qu'est parti la logique linéaire (à travers la théorie des espaces cohérents).

Le terme linéaire vient du fait qu'une preuve consomme ses hypothèses définitivement de façon linéaire. Les formules peuvent maintenant être vues comme des ressources limitées manipulées par des opérateurs plus calculatoires que logiques ($\otimes$ est simplement la juxtaposition de formules, on peut aller plus loin et parler d'opérations topologiques sur les preuves).

Globalement, c'est un système logique où l'on dispose d'un contrôle plus fin tout en permettant de reconstituer la logique usuelle.

3.3 λ-calcul affine exponentiel

Par l'isomorphisme de Curry-Howard, tout système logique cohérent induit un système de programmation. La logique linéaire induit donc une décomposition du λ-calcul.

Pour avoir plus de liberté tout en conservant des résultats satisfaisants, dans nos travaux, nous choisissons de travailler avec un calcul nommé λ-calcul *affine exponentiel* $\Lambda_!^{\mathtt{aff}}$ découlant d'une variante de la logique linéaire où l'on autorise la décomposition linéaire-exponentiel et la présence de la règle d'effacement (donc chaque ressource est consommée au plus une fois).

Definition 17 (calcul $\Lambda_!^{\mathtt{aff}}$).

Sans entrer dans les détails [a], cela induit le système de termes suivant :

$$M, N := a \mid \lambda a.M \mid MN \mid x \mid !M \mid M[!x := N]$$

Quelques points sont à noter :

- a, b, c sont les variables affines ne pouvant être utilisées qu'au plus une fois
- x, y, z sont les variables exponentielles pouvant être utilisées autant de fois que voulues
- Pour les fonctions, a apparaît au plus une fois dans M
- $!M$ (appelé *boîte*) est un terme utilisable autant de fois que l'on veut et ne contient pas d'occurrences libres de variables affines (c'est à dire non associée à un λ)
- $M[!x := N]$ (appelé `let`) représente la substitution de tous les x dans M par N sous forme de terme. [b]

a. L'émergence d'un tel calcul est décrit dans [Maz17b]

b. C'est en fait l'expression `let x = N in M` en OCaml.

Voici quelques exemples illustratifs où M et N sont des termes quelconques de $\Lambda_!^{\mathtt{aff}}$:

Termes valides	**Termes invalides**
$\lambda a.xy$	$\lambda a.Maa$
$M(!x)$	$!(\lambda a.b)$
$\lambda a.(axx)[!x := b]$	

Definition 18 (réduction de $\Lambda_!^{\mathtt{aff}}$).

$$(\lambda a.M)[-]N \longrightarrow M\{a := N\} \qquad M[!x := N[-]] \longrightarrow M\{x := N\}[-]$$
$$\text{où} \quad [-] := [-] \mid [-][!x := M]$$

On constate qu'il est possible de faire des réductions *à distance* à travers plusieurs `let`. Comme la substitution devient interne au termes on obtient plusieurs termes différents ayant

la même signification opérationnelle. On doit ignorer la distance que créer le terme de substitution. Un tel mécanisme peut être retrouvé dans le λ-calcul structurel d'Accattoli et Kesner [AK10].

Definition 19 (profondeur exponentielle).

La profondeur exponentielle d'un terme M de $\Lambda_{!\texttt{aff}}$ est définie par :

$$expdepth(a) = 0 \qquad expdepth(x) = 0 \qquad expdepth(\lambda a.M) = expdepth(M)$$

$$expdepth(MN) = expdepth(M[!x := N]) = max(expdepth(M), expdepth(N))$$

$$expdepth(!M) = expdepth(M) + 1$$

Definition 20 (décomposition de Λ).

Comme dit précédemment, $\Lambda_!^{\texttt{aff}}$ n'est qu'une décomposition de Λ. Plus précisément, cette décomposition est définie à travers la *traduction de Girard* :

$$[\![x]\!]_G = x \qquad [\![\lambda x.M]\!]_G = \lambda a.[\![M]\!]_G[!x := a] \qquad [\![MN]\!]_G = [\![M]\!]_G ![\![N]\!]_G$$

3.4 Approximations polyadiques

Lorsque Jean-Yves Girard a introduit la logique linéaire [Gir87], il a laissé une petite intuition très intéressante (page 92-93) sur ce qu'il a appelé « théorème d'approximation ». Par simplicité, nous n'exposons qu'une partie du théorème pour pouvoir capturer son essence.

Definition 21 (approximant).

On peut approximer le connecteur ! par un connecteur $!_n$ avec $n \in \mathbb{N}^*$ défini par

$$!_n A = (1\&A) \underbrace{\otimes \quad \cdots \quad \otimes}_{\text{n}} (1\&A)$$

On remplace en fait $!A$ (utilisation illimitée) par un nombre précis de ressources utilisables. La formule $(1\&A)$ représente un choix entre "rien" et A. On autorise donc le fait d'avoir plus de ressources que nécessaire.

Théorème 1 (théorème d'approximation).

Soit A une formule prouvable de la logique linéaire. Pour chaque occurrence de ! dans A il existe un entier $n \in \mathbb{N}^*$ tel que $!_n A$ soit prouvable.

Ce théorème montre que le fragment purement linéaire de la logique linéaire approxime en quelque sorte la logique linéaire complète (avec exponentielles). Quelques travaux ont tenté d'y donner une interprétation technique comme ceux de Mazza [Maz13, Maz12] qui applique cette idée sur les termes afin de donner naissance à un λ-calcul infinitaire Λ_∞ qui est un λ-calcul où l'usage infini des ressources est explicite :

— $M \in \Lambda_!^{\mathtt{aff}}$ devient $\langle M, M, ... \rangle$
— $M[!x := N] \in \Lambda_!^{\mathtt{aff}}$ devient $M[\langle x, x, ... \rangle := \langle N, N, ... \rangle]$

Cette variante, isomorphe à Λ [Maz12] est obtenue par complétion métrique de l'espace des λ-termes affines [Maz13]. En extrayant la partie finie de Λ_∞ on se retrouve avec encore une fois un nouveau calcul nommé λ-calcul polyadique affine $\Lambda_{\mathtt{p}}^{\mathtt{aff}}$ qui spécifie explicitement un nombre de ressources disponibles. C'est exactement la construction simple à maîtriser que l'on cherchait depuis le début. On a finalement la correspondance suivante :

	Système	**Approximé par**
Formules	Logique Linéaire complète (avec !)	Logique purement linéaire (sans !)
Termes	$\Lambda \cong \Lambda_\infty$	$\Lambda_{\mathtt{p}}^{\mathtt{aff}}$

C'est exactement le théorème d'approximation mais sous le point de vue des termes plutôt que des formules.

Definition 22 (calcul $\Lambda_{\mathtt{p}}^{\mathtt{aff}}$).

Les termes sont définis par :

$$t, u := a \mid \lambda a.t \mid tu \mid x \mid \langle t_1, ..., t_n \rangle \mid t[\langle x_1, ..., x_n \rangle := u]$$

Definition 23 (réduction de $\Lambda_{\mathtt{p}}^{\mathtt{aff}}$).

$$(\lambda a.t)u \to t\{a := u\}$$

$$t[-][\langle x_1, ..., x_n \rangle := \langle u_1, ..., u_m \rangle] \to t\{x_1 := u_1, ..., x_n := u_n\} \qquad \text{si } n \leq m$$

où $[-]$ est une séquence de $[\langle x_1, ..., x_n \rangle := \langle u_1, ..., u_m \rangle]$.

On voit que la réduction ne fait que connecter des occurrences unes à unes. C'est une version très statique et limitée du λ-calcul. En particulier, si la connexion n'est pas cohérente alors la réduction n'est pas définie.

On peut maintenant définir comment un terme $t \in \Lambda_{\mathtt{p}}^{\mathtt{aff}}$ approxime un terme $M \in \Lambda_!^{\mathtt{aff}}$, conformément au théorème d'approximation :

Definition 24 (approximation polyadique).

$$\frac{}{\vdash a \sqsubset a}\ lvar \qquad \frac{\Gamma \vdash t \sqsubset M}{\Gamma \vdash \lambda a.t \sqsubset \lambda a.M}\ lam$$

$$\frac{\Gamma \vdash t \sqsubset M \qquad \Delta \vdash u \sqsubset N}{\Gamma, \Delta \vdash tu \sqsubset MN}\ app \qquad \frac{\Gamma_1 \vdash t_1 \sqsubset M \quad \cdots \quad \Gamma_n \vdash t_n \sqsubset M}{\Gamma_1, ..., \Gamma_n \vdash \langle t_1, ..., t_n \rangle \sqsubset !M}\ !$$

$$\frac{}{\Gamma, x_0 \sqsubset x \vdash x_0 \sqsubset x}\ cvar \qquad \frac{\Delta \vdash u \sqsubset N \qquad \Gamma, x_1 \sqsubset x, ..., x_n \sqsubset x \vdash t \sqsubset M}{\Gamma, \Delta \vdash t[\langle x_1, ..., x_n \rangle := u] \sqsubset M[!x := N]}\ let$$

Comme pour les règles logiques, les règles d'approximations sont à lire du haut vers le bas (hypothèses vers conclusion). Une expression de la forme $\Gamma \vdash t \sqsubset M$ signifie que le terme t approxime M sachant qu'on connaît les approximations contenues dans Γ (contexte d'approximation).

On peut aussi définir une relation d'approximation entre les termes polyadiques, induisant une relation d'ordre fondée sur le nombre de ressources disponibles.

Definition 25 (ordre d'approximation polyadique).

$$\frac{}{a \sqsubseteq a}\ lvar \qquad \frac{t \sqsubseteq t'}{\lambda a.t \sqsubseteq \lambda a.t'}\ lam \qquad \frac{t \sqsubseteq t' \qquad u \sqsubseteq u'}{tu \sqsubseteq t'u'}\ app \qquad \frac{}{x \sqsubseteq x}\ cvar$$

$$\frac{t_1 \sqsubseteq u_1 \quad \cdots \quad t_n \sqsubseteq u_n}{\langle t_1, ..., t_n \rangle \sqsubseteq \langle u_1, ..., u_m \rangle}\ !n \leq m \qquad \frac{t \sqsubseteq t' \qquad u \sqsubseteq u' \qquad x_{n+1}, ..., x_m \notin fv(t)}{t[\langle x_1, ..., x_n \rangle := u] \sqsubseteq t'[\langle x_1, ..., x_m \rangle := u']}\ let\ n \leq m$$

Deux propriétés primordiales justifient l'utilité des approximations polyadiques [Maz17b] :

Propriété 2 (continuité de la réduction).

Soit $M, N \in \Lambda_!^{\mathtt{aff}}$ et $t, u \in \Lambda_{\mathtt{p}}^{\mathtt{aff}}$.
Si on a

$$M \longrightarrow^* N \quad \text{et} \quad u \sqsubset N$$

alors il existe t tel que

$$t \sqsubset M \quad \text{et} \quad t \longrightarrow^* u$$

C'est à dire que la réduction dans $\Lambda_!^{\mathtt{aff}}$ entre termes à ressources infinies peut aussi être approximé dans $\Lambda_{\mathtt{p}}^{\mathtt{aff}}$ par un terme affine aux ressources finies et statiques.

Propriété 3 (monotonicité de la réduction).

Soit $t, t', u, u' \in \Lambda_{\mathtt{p}}^{\mathtt{aff}}$.
Si on a

$$t \to u \quad \text{et} \quad t \sqsubseteq t'$$

alors il existe u' tel que

$$t' \to u' \quad \text{et} \quad u \sqsubseteq u'$$

3.5 Transformations de termes

Quelques fonctions de transformation nous seront utiles afin de construire nos propres approximations polyadiques affines.

Definition 26 (approximation homogène).

L'approximation homogène d'un λ-terme $M \in \Lambda_!^{\mathtt{aff}}$ est définie par :

$$\lfloor a \rfloor_k = a \qquad \lfloor \lambda a.M \rfloor_k = \lambda a.\lfloor M \rfloor_k \qquad \lfloor MN \rfloor_k = \lfloor M \rfloor_k \lfloor N \rfloor_k$$

$$\lfloor x \rfloor_k = x \qquad \lfloor M[!x := N] \rfloor_k = \lfloor M \rfloor_k [\langle x \rangle := \lfloor N \rfloor_k] \qquad \lfloor !M \rfloor_k = \langle \underbrace{\lfloor M \rfloor_k, ..., \lfloor M \rfloor_k}_{k} \rangle$$

La fonction ci-dessus permet de forcer la borne des ressources du calcul à un certain entier $k \in \mathbb{N}$. Le retour de la fonction est un calcul polyadique non-affine qu'on ne décrit pas.

Definition 27 (linéarisation).

La linéarisation $lin(t)$ d'un λ-terme t polyadique non-affine consiste à rendre unique toutes les occurrences de variables. On définit la linéarisation d'adresse initiale ι ainsi :

$$lin_\iota(a) = a \qquad lin_\iota(tu) = lin_{\iota \cdot 1}(t) lin_{\iota \cdot 2}(u) \qquad lin_\iota(\lambda a.t) = \lambda a.lin_\iota(t)$$

$$lin_\iota(x) = x_\iota \qquad lin_\iota(\langle t_1, ..., t_n \rangle) = \langle lin_{\iota \cdot 1}(t_1), ..., lin_{\iota \cdot n}(t_n) \rangle$$

$$lin_\iota(t[\langle x \rangle := u]) = lin_\iota(t)\{x_\_ :=\}[\langle x_1, ..., x_n \rangle := u] \qquad lin(t) = lin_\varepsilon(t)$$

Cette fonction retourne un terme de $\Lambda_{\mathtt{p}}^{\mathtt{aff}}$ et sera souvent utilisée de paire avec l'approximation homogène pour pouvoir produire des approximations polyadiques.

4 Outils logiques

Maintenant qu'on dispose du calcul $\Lambda_{\mathtt{p}}^{\mathtt{aff}}$ au mécanisme fini et précis, nous pouvons introduire les outils qui vont nous permettre de mesurer convenablement l'espace du calcul dans les termes.

4.1 Types intersections

Les systèmes de types intersections introduits par [CD80] sont des systèmes de type permettant de capturer la classe des termes qui terminent grâce à un nouveau type $M : \langle A_1, ..., A_n \rangle$ pouvant donner plusieurs types à M [1]. Mais son intérêt ne s'arrête pas ici, si on travaille sur un système « non-idempotent » [Gar94, DC17] le type des termes contient en fait des informations quantitatives importantes sur le calcul. C'est à dire qu'au lieu d'avoir

$$A \equiv \langle A, A \rangle$$

qui cache et écrase la mesure du calcul on a

$$A \not\equiv \langle A, A \rangle$$

1. C'est en fait une forme de polymorphisme

Cette inégalité va permettre, en quelque sorte, de tracer l'usage des occurrences dans le calcul. Si un terme M utilise n occurrences de termes N de type A alors le type de M contiendra l'intersection $\langle \underbrace{A, ..., A}_{\text{n}} \rangle$.

L'inférence de type étant indécidable, l'usage d'un tel système n'est que purement théorique. Cependant le *typechecking* est décidable : on peut définir une procédure qui vérifie si un type A donné permet bien de typer un terme M. Le système de type intersection que nous définissons associera un type aux termes de $\Lambda_!^{\mathtt{aff}}$.

Definition 28 (types intersections).

La grammaire des types est

$$A, B := \alpha \mid A \multimap B \mid \langle A_1, ..., A_n \rangle$$

Les règles de typage sont les suivantes où $\Gamma \vdash M : A$ signifie que le terme M est associé au type A avec l'environnement de typage Γ :

$$\frac{}{\Theta \mid \Gamma, a : A \vdash a : A}\ lvar \qquad \frac{\Theta \mid \Gamma, a : A \vdash M : B}{\Theta \mid \Gamma \vdash \lambda a.M : A \multimap B}\ lam$$

$$\frac{\Theta \mid \Gamma \vdash M : A \multimap B \qquad \Theta' \mid \Gamma' \vdash N : A}{\Theta \cdot \Theta' \mid \Gamma, \Gamma' \vdash MN : B}\ app \qquad \frac{\Theta_1 \mid \vdash M : A_1 \quad \cdots \quad \Theta_n \mid \vdash M : A_n}{\Theta_1 \cdot ... \cdot \Theta_n \mid \vdash !M : \langle A_1, ..., A_n \rangle}\ !$$

$$\frac{}{\Theta, x : A \mid \Gamma \vdash x : A}\ cvar \qquad \frac{\Theta \mid \Gamma, x : \langle A_1, ..., A_n \rangle \vdash M : B \qquad \Theta' \mid \Gamma' \vdash N : \langle A_1, ..., A_n \rangle}{\Theta \cdot \Theta' \mid \Gamma, \Gamma' \vdash M[!x := N] : B}\ let$$

$$\frac{\Theta, x : \langle A_1, ..., A_n \rangle \mid \Gamma \vdash M : A}{\Theta, x : \langle A_{\sigma(1)}, ..., A_{\sigma(n)} \rangle \mid \Gamma \vdash M : A}\ ex_\sigma$$

Respectant les conventions suivantes :

Séparation. On sépare le contexte en deux parties $\Theta \mid \Gamma$ où Θ est l'environnement des variables exponentielles (utilisables de façon illimitée) et Γ celui des variables linéaires (consommées à chaque usage).

Complétion. Les intersections vides sont notées $\top$. Si dans un contexte cartésien Θ, une variable x n'apparaît pas on peut le compléter avec $\Theta, x : \top$.

Concaténation. La concaténation $\Theta \cdot \Theta'$ de deux contextes cartésiens unifie toutes variables $x : \langle A_1, ..., A_n \rangle \in \Theta$ et $x : \langle B_1, ..., B_n \rangle \in \Theta'$ en $x : \langle A_1, ..., A_n, B_1, ..., B_n \rangle \in \Theta \cdot \Theta'$ et où $\top$ agit comme l'élément neutre de la concaténation.

On vérifie le type d'un terme grâce à la construction arborescente suivante appelée *arbre de dérivation de type* :

$$\dfrac{\dfrac{x : A \multimap B \mid \vdash x : A \multimap B \qquad x : A \mid \vdash x : A}{x : \langle A \multimap B, A \rangle \mid \vdash xx : B}}{\vdash \lambda x.xx : \langle A \multimap B, A \rangle \multimap B}$$

La variable x ayant à la fois rôle de fonction et de paramètre de fonction, il faut prendre ces deux usages en compte. Si un tel arbre est constructible alors on dit que le terme racine est typable dans le système de type intersection.

Definition 29 (taille de dérivation).

> La taille $size(\delta)$ ou $|\delta|$ d'une dérivation $\delta \rhd M : A$ est définie par le nombre de règles utilisées dans δ.

Definition 30 (profondeur de dérivation).

> La profondeur $depth(\delta)$ d'une dérivation $\delta \rhd M : A$ est définie par la profondeur de la formule la plus profonde utilisée dans δ. La profondeur d'une formule est définie par :
>
> $$depth(\alpha) = 1 \qquad depth(A \multimap B) = max(depth(A), depth(B)) + 1$$
>
> $$depth(\langle A_1, ..., A_n \rangle) = \left(\max_{1 \leq i \leq n} depth(A_i)\right) + log\ n$$

Dans le cadre des systèmes de type intersection, la profondeur et la taille des dérivations de types contiennent des informations importantes sur la complexité des programmes.

Lemme 4 (normalisation forte).

> Pour tout $M \in \Lambda_!^{\mathtt{aff}}$ et type A, si $\vdash M : A$ alors toute chaîne de réduction à partir de M est finie.

Lemme 5 (réduction du sujet).

> Pour tout $M \in \Lambda_!^{\mathtt{aff}}$ et type A tel que $\vdash M : A$, s'il existe N tel que $M \longrightarrow^* N$ alors $N : A$. En d'autres termes, le type est préservé au cours de la réduction.

Ce système de type intersection est en fait plus proche de la logique linéaire et des approximations polyadiques qu'on ne le croit : le type intersection d'un terme peut être vu comme le typage simple d'un terme polyadique affine l'approximant. Ce constat induit le lemme suivant :

Lemme 6 (terme polyadique affine induit).

> Tout arbre de dérivation de système de type intersection $\delta \rhd M : A$ induit un terme polyadique affine δ^- tel que $\delta^- \sqsubset M$.

Mazza, Pellissier et Vial font l'analyse d'une telle correspondance dans un cadre catégorique (théorie des opérades) [Maz17a].

4.2 Géométrie de l'interaction

La géométrie de l'interaction est l'outil phare derrière notre mesure de l'espace. Au début de cet article nous avons vu qu'une mesure naïve de l'espace des termes ne fonctionnait pas. Nous avons maintenant tout en main pour résoudre ce problème.

La géométrie de l'interaction [Gir89] représente un ensemble de méthodes issues de la logique linéaire permettant d'exécuter un terme du λ-calcul linéaire ou affine sans aucun binding variable-valeur ni réécriture syntaxique. On exécute le terme de façon statique.

L'intuition est qu'un terme de $\Lambda_{\mathtt{p}}^{\mathtt{aff}}$ peut-être vu comme une preuve de la logique linéaire. Ensuite, la théorie de la géométrie de l'interaction nous dit qu'on peut caractériser une preuve de la logique linéaire par ses chemins internes. En conclusion, parcourir les chemins internes d'un terme affine ou linéaire permet donc d'avoir des informations sur le résultat de sa réduction.

Nous introduisons la géométrie de l'interaction par une machine abstraite appelée *machine à jeton* ou *machine interactive abstraite* malgré qu'il existe plein d'autres formulations (dans les catégories monoïdales tracées, les algèbres stellaires, les réseaux de preuve, etc)

Definition 31 (configuration).

> Une configuration est un quadruplet $(d \mid t \mid C \mid S)$ où
> - $d \in \{\overline{\wedge}, \underline{\vee}\}$ est la direction
> - $t \in \Lambda_{\mathtt{p}}^{\mathtt{aff}}$ est le terme polyadique affine en entrée
> - C, appelé contexte, est un terme polyadique affine avec un unique trou $\{\cdot\}$
> - S est une pile d'éléments de $\{p, q\} \cup \mathbb{N}$

Definition 32 (règles de transition).

> Les règles de transition sont données par une transformation d'une configuration à une autre décrite par le tableau de la figure 1.
>
> On définit la relation de transition $\rightsquigarrow$ par l'union des relations définies dans la même figure. La clôture transitive réflexive est notée $\rightsquigarrow^*$.
>
> L'intuition de ces règles et que l'on simule une sorte de pointeur en se focalisant sur un sous terme disponible mais qu'on garde le "reste du terme" grâce au contexte. Selon notre parcours on gère une pile qui va permettre de tracer le chemin et ainsi nous donner des informations sur l'exécutions du terme, le tout sans le réduire explicitement.

Remarque. Pour toute transition $(d \mid t \mid C \mid S) \rightsquigarrow (d' \mid t' \mid C' \mid S')$, il existe une transition duale $(\overline{d}' \mid t' \mid C' \mid S') \rightsquigarrow (\overline{d} \mid t \mid C \mid S)$ où $\overline{(\cdot)}$ inverse les directions c'est à dire que $\underline{\vee}$ devient $\overline{\wedge}$ et $\overline{\wedge}$ devient $\underline{\vee}$.

Definition 33 (configuration initiale et finale).

$\overline{\wedge}$	a	$C\{\lambda a.C'\}$	S	$\overset{\lambda p}{\rightsquigarrow}$	$\underline{\vee}$	$\lambda a.C'\{a\}$	C	$p \cdot S$
$\underline{\vee}$	t	$C\{\lambda a.\{\cdot\}\}$	S	$\overset{\lambda q}{\rightsquigarrow}$	$\underline{\vee}$	$\lambda a.t$	C	$q \cdot S$
$\underline{\vee}$	u	$C\{t\{\cdot\}\}$	S	$\overset{@p}{\rightsquigarrow}$	$\overline{\wedge}$	t	$C\{\{\cdot\}u\}$	$p \cdot S$
$\overline{\wedge}$	tu	C	S	$\overset{@q}{\rightsquigarrow}$	$\overline{\wedge}$	t	$C\{\{\cdot\}u\}$	$q \cdot S$
$\underline{\vee}$	t	$C\{\langle\cdots\{\cdot\}_i\cdots\rangle\}$	S	$\overset{\texttt{push}\langle\rangle}{\rightsquigarrow}$	$\underline{\vee}$	$\langle\cdots\{t\}_i\cdots\rangle$	C	$i \cdot S$
$\overline{\wedge}$	$\langle t_1, ..., t_n\rangle$	C	$i \cdot S$	$\overset{\texttt{pop}\langle\rangle}{\rightsquigarrow}$	$\overline{\wedge}$	t_i	$C\{\langle t_1, ..., \{\cdot\}_i, ...t_n\rangle\}$	S
$\underline{\vee}$	u	$C\{x_i\}[\langle\vec{x}\rangle := \{\cdot\}]$	$i \cdot S$	$\overset{\texttt{pop!}}{\rightsquigarrow}$	$\underline{\vee}$	x_i	$C[\langle\vec{x}\rangle := u]$	S
$\overline{\wedge}$	x_i	$C[\langle\vec{x}\rangle := u]$	S	$\overset{\texttt{push!}}{\rightsquigarrow}$	$\overline{\wedge}$	u	$C\{x_i\}[\langle\vec{x}\rangle := \{\cdot\}]$	$i \cdot S$
$\underline{\vee}$	t	$C\{\{\cdot\}[\langle\vec{x}\rangle := u]\}$	S	$\overset{\texttt{C!}}{\rightsquigarrow}$	$\underline{\vee}$	$t[\langle\vec{x}\rangle := u]$	C	S

FIGURE 1 – Transitions de la machine à jetons

La notation $\langle\cdots\{t\}_i\cdots\rangle$ signifie que l'on se focalise sur la position i d'une séquence polyadique quelconque pour y insérer le terme t où le contexte vide dans le cas de $\{\cdot\}_i$.

Pour toute pile S on définit les configurations suivantes appelées initiale et finale :
- $init(t, S) := (\overline{\wedge} \mid t \mid \{\cdot\} \mid S)$
- $fin(t, S) := (\underline{\vee} \mid t \mid \{\cdot\} \mid S)$

Propriété 7 (bi-déterminisme).

Pour chaque configuration c,
- S'il existe une configuration c' tel que $c \rightsquigarrow c'$ alors elle est unique.
- S'il existe une configuration c' tel que $c' \rightsquigarrow c$ alors elle est unique.

Definition 34 (exécution).

On définit une relation binaire d'exécution $S \overset{t}{\leftrightsquigarrow} S'$ si $init(t, S) \rightsquigarrow^* fin(t, S')$ représentant l'exécution d'un terme t d'une pile S à une pile S'.

Propriété 8 (stabilité des contextes).

$(d \mid t \mid C \mid S) \rightsquigarrow (d' \mid t' \mid C' \mid S')$ implique $C\{t\} = C'\{t'\}$. Cette propriété montre que la notion de contexte ne permet que de simuler une notion de focalisation (pointeur) sur un terme mais que celui-ci reste inchangé au cours des transitions.

Definition 35 (ordre sur les configurations).

On peut étendre l'ordre d'approximation aux configurations par

$$(d \mid t \mid C \mid S) \sqsubseteq (d' \mid t' \mid C' \mid S') \quad \text{si et seulement si} \quad t \sqsubseteq t', \quad C \sqsubseteq C', \quad d = d', \quad S = S'$$

Sachant qu'on a $\{\cdot\} \sqsubseteq \{\cdot\}$ (un contexte vide ne contient pas de séquence polyadique qu'on peut étendre). Si $c_1 \sqsubseteq c_2$ on dit que c_2 est une **extension** de c_1.

On définit maintenant le théorème de monotonicité énonçant que le fonctionnement de la machine à jeton sur un terme $t \in \Lambda_{\mathtt{p}}^{\mathtt{aff}}$ est le même sur une de ses sur-approximations.

Théorème 9 (monotonicité de la transition).

Si $c_1 \rightsquigarrow c_2$ et $c_1 \sqsubseteq c_1'$ alors il existe c_2' tel que $c_1' \rightsquigarrow c_2'$ et $c_2 \sqsubseteq c_2'$. On peut représenter la propriété par le diagramme suivant :

$$\begin{array}{ccc} c_1' = (d \mid \lceil t \rceil \mid \lceil C \rceil \mid S) & \overset{\exists ?}{\dashrightarrow} & c_2' = (d' \mid \lceil t' \rceil \mid \lceil C' \rceil \mid S') \\ \sqsubseteq \uparrow & & \uparrow \sqsubseteq \\ c_1 = (d \mid t \mid C \mid S) & \rightsquigarrow & c_2 = (d' \mid t' \mid C' \mid S') \end{array}$$

où $\lceil t \rceil$ est une sur-approximation de t. C'est à dire qu'on a forcément $t \sqsubseteq \lceil t \rceil$.

Preuve.

Par cas sur $c_1 \rightsquigarrow c_2$. On ne se concentre que sur les cas où l'extension de séquence polyadique est visible. On voit sur le diagramme que l'ordre sur les configurations n'affectant par la direction et la pile, il suffit d'utiliser l'extension qui a permis de passer de c_1 à c_1' et de l'appliquer à c_2.

— Soit $c_1 = (\underline{\vee} \mid t \mid C\{\langle t_1, ..., \{\cdot\}_i, ..., t_n\rangle\} \mid S) \overset{\mathtt{push}\langle\rangle}{\rightsquigarrow} c_2 = (\underline{\vee} \mid \langle t_1, ..., \{t\}_i, ..., t_n\rangle \mid C \mid i \cdot S)$. On sait que c_1 est étendu à un certain

$$c_1' = (\underline{\vee} \mid t \mid C\{\langle t_1, ..., \{\cdot\}_i, ..., t_n, t_{n+1}, ..., t_m\rangle\} \mid S)$$

Si on étend c_2 de la même façon on a

$$c_2' = (\underline{\vee} \mid \langle t_1, ..., \{t\}_i, ..., t_n, t_{n+1}, ..., t_m\rangle \mid C \mid i \cdot S)$$

La transition $c_1 \rightsquigarrow c_2$ implique que l'indice i est hors de l'extension, c'est à dire $1 \leq i \leq n$. On a donc $c_1' \overset{\mathtt{push}\langle\rangle}{\rightsquigarrow} c_2'$ et $c_2 \sqsubseteq c_2'$ par construction.

— Les autres cas $\overset{\mathtt{pop}\langle\rangle}{\rightsquigarrow}$, $\overset{\mathtt{pop!}}{\rightsquigarrow}$ et $\overset{\mathtt{c!}}{\rightsquigarrow}$ suivent un raisonnement similaire qu'on peut aisément vérifier. Il suffit de vérifier que l'extension n'affecte pas la transition.

— Dans les autres cas, puisqu'il y n'y a pas d'extension de séquence polyadique, la même transition reste valide. C'est à dire que l'on a $c_2' = c_2$.

□

Corollaire 10 (monotonicité de l'exécution).

Si $t \sqsubseteq t'$ alors $\overset{t}{\leftsquigarrow} \subseteq \overset{t'}{\leftsquigarrow}$.

Preuve.

Trivialement, par le théorème précédent, l'exécution de la machine à jeton sur t a le même comportement que sur t'. Le raisonnement se fait par récurrence sur la longueur de l'exécution $init(t, S) \rightsquigarrow^* fin(t, S')$ pour des piles S et S' quelconques.

□

5 Résultats

Il est maintenant temps de faire appel à tous nos outils. Nous énonçons à titre anecdotique une motivation secondaire de ce travail puis donnons une simulation en espace constant d'un terme de $\Lambda_!^{\mathtt{aff}}$ dans les machines de Turing.

5.1 La motivation des circuits

En termes profanes (une définition formelle est proposée dans l'annexe), les circuits booléens sont une généralisation des formules booléennes. On peut les voir comme un graphe représentant une formule booléenne où il y a éventuellement un partage de sous-formules.

Une analogie proposée dans plusieurs écrits de Damiano Mazza [Maz17b, Mazb, Maza] propose de voir les λ-terme linéaires ou affines comme des *circuits booléens d'ordre supérieur*. Terme qui peut s'expliquer par les points suivants :

— comme les circuits, les termes affines peuvent résoudre des problèmes finis. Par contre, contrairement aux circuits ils peuvent aussi traiter d'autres termes (d'où l'appellation "ordre supérieur").

— les deux ont des mécanismes très similaires. Pour résoudre un problème ils connectent l'entrée et la sortie de façon statique.

— il est connu dans la littérature qu'une machine de Turing peut être approximée par une famille potentiellement infinie de circuits booléens. On sait qu'une telle approximation existe pour les termes.

La correspondance suivante émerge :

$$\frac{\text{Machine de Turing}}{\text{Circuit booléen}} \cong \frac{\lambda\text{-calcul linéaire exponentiel}}{\lambda\text{-calcul polyadique affine}}$$

Cette correspondance n'est pas nouvelle puisqu'auparavant, Mairson [Mai04] avait déjà montré une correspondance très simple entre λ-termes affines et circuits booléens. S'en est suivi une correspondance entre circuits booléens et preuves de la logique linéaire par Terui [Ter04] rendant encore plus étroite les corrélations entre logique linéaire, λ-calcul, circuits booléens et machines de Turing.

En 1977, Borodin montre que $\mathbf{DEPTH}(f) \subseteq \mathbf{SPACE}(O(f))$ [Bor77], c'est-à-dire qu'une famille de circuits de profondeur $f(n)$ peut être simulée par une machine de Turing en temps constant $O(f(n))$. Avec la correspondance ci-dessus en tête nous essayons de reproduire ce résultat dans le monde des termes.

5.2 Classes de complexité

Nous introduisons les classes de complexité qui nous seront utiles pour la suite.

— $\lambda\mathbf{DEPTH\text{-}SIZE}(f, g)$ est la classe des problèmes décidables par un terme de $\Lambda_!^{\mathtt{aff}}$ typé par une dérivation de profondeur $f(n)$ et de taille $g(n)$ dans un système de type intersection.

— $\mathbf{SPACE}(f)$ est la classe des problèmes décidables par une machine de Turing déterministe en espace $f(n)$.

5.3 Conventions de types

Plaçons-nous dans le contexte des types intersections pour le calcul $\Lambda_!^{\mathtt{aff}}$. Les types que nous utilisons sont les suivants où X est un atome (type de base) quelconque :

$$Bool \coloneqq X \multimap X \multimap X$$

$$Str_n \coloneqq \langle X_0^n \multimap X_1^n, ..., X_{n-1}^n \multimap X_n^n \rangle \multimap \langle X_0^n \to X_1^n, ..., X_{n-1}^n \multimap X_n^n \rangle \multimap X_0^n \multimap X_n^n$$

On définit le raccourci $Str_n[\overrightarrow{B^n}]$ pour $Str_n\{X_1 \coloneqq B_0\}\cdots\{X_n \coloneqq B_n\}$ avec $\overrightarrow{B^n}$ une séquence finie de types. Cette technicité est expliquée dans la conclusion.

Dans $\Lambda_!^{\mathtt{aff}}$, le codage des chaînes binaires devient :

$$\underline{w_1...w_n \in \{0,1\}^n} = \lambda s_0.\lambda s_1.\lambda e.\underbrace{x_{w_1}(...(x_{w_n}}_{\text{n}}\ e)...)[!x_0 \coloneqq s_0][!x_1 \coloneqq s_1]$$

ε est l'unique mot binaire de taille 0 dont le codage est $\underline{\varepsilon} = \lambda s_0.\lambda s_1.\lambda e.e$

5.4 Théorème de borne spatiale

Voici le théorème principal qui va borner la complexité en espace d'un décideur de $\Lambda_!^{\mathtt{aff}}$.

Théorème 11 (borne spatiale du λ-calcul).

Soit M un terme de $\Lambda_!^{\mathtt{aff}}$ tel qu'il existe :

1. Une suite de types $(B_i^n)_{n\in\mathbb{N},0\leq i\leq n}$
2. Une suite de dérivations $\delta_n \rhd \vdash M : Str_n[\overrightarrow{B^n}] \multimap Bool$

On peut en déduire que M décide

$$L \in \mathrm{SPACE}\big(O(depth(\delta_n) + log\ rank(\delta_n))\big)$$

sous la condition que $|\delta_n| \overset{\Omega}{=} \Omega(n)$ et que $\kappa(n)$ est log-constructible en espace où $rank(\delta_n)$ est la longueur maximale des suites polyadiques des types contenus dans δ_n et $\kappa(n) \coloneqq rank(\delta_n)$. Ces procédures sont intégrées à la machine à jeton par des associations position-entier.

Preuve.

1. On commence par montrer que M décide bien un langage.
   - Soit $w \in \{0,1\}^n$. Par le lemme de typage des chaînes binaires (voir Annexe) on a $\vdash \underline{w} : Str_n[\overrightarrow{B^n}]$.
   - Par notre suite de dérivations δ_n on a donc $\vdash M\ \underline{w} : Bool$.
   - Par normalisation forte des termes typés par intersections, on a $M\ \underline{w} \longrightarrow^* \underline{b}$.
   - Par réduction du sujet, $\underline{b} : Bool$. Donc M décide bien un langage.
2. Il nous reste à borner la complexité de la décision par application de la *géométrie de l'interaction* sur un terme adéquat (linéaire ou affine) ayant le même comportement que M. Comment produire ce terme ?
   (a) Pour le lemme de terme polyadique induit, la famille de dérivations (δ_n) induit une famille de termes polyadiques affines (δ_n^-) mimant le comportement de M. Cependant, ne connaissant pas la forme des (δ_n) nous n'avons pour certitude que l'existence d'un δ_n^-.
   (b) Étant dans une impasse, on doit contourner le problème en générant un terme $t_n \in \Lambda_{\mathtt{p}}^{\mathtt{aff}}$ d'une autre manière de telle sorte qu'on ait

   $$(\delta_n^-) \sqsubseteq (t_n) \sqsubset M$$

   C'est à dire que l'on créer un terme artificiel qui est M mais auquel on donne une limite explicite de ressources utilisables sur toutes les séquences polyadiques.
   (c) Le maximum de ressources utilisées est en fait borné par le rang $\kappa(n)$ (on rappelle que cette information est contenue dans la dérivation de type intersection qui trace l'usage des occurrences). On va donc créer un terme disposant de plus de ressources qu'il en a réellement besoin mais qui en a nécessairement assez. Pour ce faire, on passe par les *approximations homogènes* $\lfloor M \rfloor_{\kappa(n)}$. N'ayant pas encore un terme linéaire ou affine à ce stade, il faut appliquer une procédure de

linéarisation en rendant toutes les occurrences de variables uniques. On obtient le terme suivant traitable par la géométrie de l'interaction :

$$t_n \coloneqq lin(\lfloor M \rfloor_{\kappa(n)})$$

On a ensuite la propriété que t_n simule le comportement de M par *monotonicité* et *continuité* de la réduction. Le lemme de négligence spatiale (voir Annexe) et la supposition que $\kappa(n)$ est logspace-constructible nous assurent que la perte en espace due à ces procédures est négligeable.

(d) Les suites polyadiques de t_n sont de taille supérieure ou égale à celles de M et donc à celles de δ_n^- par construction. Les deux étant des approximations de M, donc on a bien

$$(\delta_n^-) \sqsubseteq (t_n) \sqsubset M$$

(e) Le lemme de monotonicité de l'exécution nous assure que la machine à jeton lancé sur t_n a le même comportement que sur δ_n^- car l'exécution théoriquement valide sur δ_n^- existe toujours dans la sur-approximation t_n. L'extension des séquences polyadiques n'est pas explorée.

(f) On lance une *machine à jeton* sur t_n. L'exécution nécessite un *pointeur* de taille

$$log|t_n w_n^-| \quad = \quad log(O(|\delta_n| + n)) \quad \overset{\Omega}{=} \quad O(log|\delta_n|)$$

pour explorer le terme, une pile contenant des valeurs de $\{\mathtt{p}, \mathtt{q}\} \cup \mathbb{N}$ dont la longueur est bornée par $depth(Str_n[\overrightarrow{B^n}]) = depth(\delta_n)$ pour enregistrer des informations sur l'exécution.

En composant l'espace nécessaire à chaque procédure, il est évident que la borne en espace énoncée est une borne supérieure.

□

Le résultat ci-dessus est en fait le parfait analogue du résultat de Borodin [Bor77] simulant une famille de circuits de profondeur $f(n)$ et de taille $g(n)$ en espace $O(f + \log\ g)$, dévoilant ainsi une corrélation temps/taille et espace/profondeur qui s'applique aussi dans notre contexte.

Corollaire 12 (simulation).

λ**DEPTH-SIZE**$(f, g) \subseteq$ **SPACE**$(O(f + log\ g))$

Preuve.

Montrons que s'il existe $M \in \Lambda_!^{\mathtt{aff}}$ typé par une dérivation de type intersection de profondeur $f(n)$ et de taille $g(n)$ décidant $\mathcal{L}$ alors il existe une machine $\mathfrak{D}$ décidant $\mathcal{L}$ en espace $O(f + log\ g)$. Il s'agit de simuler le comportement de M par $\mathcal{D}$. On s'abstrait de tout détail d'implémentation bas niveau en supposant qu'ils ne nous gêneront pas.

C'est une conséquence directe du théorème de borne spatiale. On peut coder la machine à jeton dans $\mathfrak{D}$. Le terme $M \in \Lambda_!^{\mathtt{aff}}$ et sa dérivation δ_n étant constants (fixés), on peut supposer que la machine sait calculer les rangs avec une procédure log-space constructible

par association de valeurs.

L'espace nécessaire à la création d'un terme t_n issu de M traitable par la machine à jeton prenant un espace $O(f + log\ rank(\delta_n))$, et $rank(\delta_n)$ étant borné par $size(\delta_n)$, on a bien $\mathcal{L}$ décidé en espace $O(f + log\ rank(\delta_n))$ par $\mathfrak{D}$.

□

Théorème 13 (inclusion inverse).

SPACE$(f) \subseteq \lambda$**DEPTH-SIZE**$(O(f^2), O(2^f))$

Preuve.

C'est en fait, une fois de plus, l'analogue d'un résultat connu des circuits booléens [Bor77]. Ce dernier résultat énonce

$$\textbf{NSPACE}(f) \subseteq \textbf{DEPTH-SIZE}O(f^2), O(2^f)$$

où **DEPTH-SIZE** est la classe des circuits booléens de profondeur polynomiale et de taille exponentielle. La classe **NSPACE**(f) induit donc une famille de circuits conforme à ce résultat. On conclut avec le codage efficace de Mairson [Mai04] des circuits booléens dans les λ-termes affines (perte d'espace logarithmique et de temps linéaire) qui sont des cas particuliers de termes de $\Lambda_!^{\mathtt{aff}}$ et le fait connu que **SPACE**$(f) \subseteq$ **NSPACE**$(O(f))$.

□

5.5 Raisonnabilité de λSPACE(f)

Est-ce que notre mesure représente un modèle de coût raisonnable ? Nous pensons que l'espace des machines coïncide avec la profondeur des dérivations de type intersection pour les termes de $\Lambda_!^{\mathtt{aff}}$ et que la taille peut être ignorée. Cela nous permet de définir une classe λ**SPACE**(f) comme étant la classe des problèmes décidables par un terme de $\Lambda_!^{\mathtt{aff}}$ typé par une dérivation de profondeur $f(n)$ dans un système de type intersection.

Un premier petit constant est de remarquer que la taille est borné par l'exponentiation de la profondeur.

Lemme 14 (borne exponentielle de la taille des dérivations).

Soit $\delta \rhd M : A$, un terme M de $\Lambda_!^{\mathtt{aff}}$ typé par une dérivation δ dans les types intersections. Si d est la profondeur de δ alors on a $|\delta| = O(2^d)$.

Preuve.

La preuve utilisant des arguments provenant de la théorie du λ-calcul et de la logique linéaire (η-expansion et réseaux de preuve) elle n'est pas détaillée ici. Elle repose fondamentalement sur des propriétés connues associant taille et profondeur des arbres n-aires.

□

La correspondance entre espace des machines et termes s'exprime ainsi par le théorème suivant (on rappelle que la simulation doit se faire en espace constant dans les deux sens) :

Corollaire 15 (raisonnabilité).

$\lambda\mathbf{SPACE}(f) = \mathbf{SPACE}(f)$

Preuve.

— $\lambda\mathbf{SPACE}(f) \subseteq \mathbf{SPACE}(O(f))$: c'est une conséquence du théorème de borne spatiale. Il faut utiliser le lemme de borne exponentielle de la taille des dérivations. Ce qui nous donne une borne de $SPACE(O(d + log\ 2^d)) = SPACE(O(d + d)) = SPACE(O(d))$ pour une dérivation de taille t et de profondeur d dans un système de type intersection.

— $\mathbf{SPACE}(f) \subseteq \lambda\mathbf{SPACE}(O(f))$: nous n'avons pas exploré cette partie. Cependant, nous en discutons dans la conclusion.

□

5.6 Uniformité des approximations

Le théorème de borne spatiale ne borne pas simplement l'espace du λ-calcul : on remarque que la preuve induit une famille de termes polyadiques affines $(t_n)_{n\in\mathbb{N}}$ approximant le terme M de départ.

Un phénomène très similaire se produit dans la preuve du théorème de Cook-Levin [Coo71] montrant que le problème **SAT** est **NP**-complet. Une lecture différente de ce théorème montre qu'en fait une machine de Turing (déterministe si on voit la machine comme un vérifieur) induit une certaine famille de circuits booléens. On dit que cette famille est uniforme c'est à dire qu'on peut la générer efficacement (avec une conception subjective d'efficacité).

Nous aimerions continuer de rapprocher la complexité des machines et des termes en montrant que la famille $(t_n)_{n\in\mathbb{N}}$ est aussi uniforme et que leur approximation de M est analogue à l'approximation d'une machine par des circuits.

Nous définissons une notion d'uniformité pour les λ-termes polyadiques affines et montrons que la famille d'approximation $(\lfloor M \rfloor_k)_{k\in\mathbb{N}}$ est uniforme et que la famille peut être générée en temps logarithmique. Commençons par quelques conventions sur le codage binaire.

Definition 36 (padding binaire).

Le padding binaire est une fonction $(\{0,1\}^* \times \mathbb{N}) \to \{0,1\}^*$ définie par

$$\texttt{padbin}(w, max) = 0^{max-|w|} \bullet w$$

permettant de combler un nombre binaire par des zéros inutiles devant. Cela permettra de normaliser les entiers binaires à une même longueur.

Definition 37 (codage unaire et binaire).

$\texttt{una}(n)$ et $\texttt{bin}(n)$ sont respectivement le codage unaire et binaire de l'entier n.

Definition 38 (codage binaire normalisé).

$\texttt{bin}_k(n) = \texttt{padbin}(\texttt{bin}(n), k)$ est le codage normalisé de l'entier n.

On s'inspire maintenant d'une description des circuits appelée *langage de connexion directe* [Vol99] que l'on adapte à $\Lambda_{\texttt{p}}^{\texttt{aff}}$. Ce langage permet de représenter une famille de termes par un ensemble. Un décideur pour ce langage vérifie si l'entrée se réfère à une position existante dans le terme d'un des membres de la famille.

Definition 39 (adressage approximatif des termes polyadiques affines).

Sachant un r donné, on associe une adresse à un sous-terme u de $t \in \Lambda_{\texttt{p}}^{\texttt{aff}}$ par la fonction $\texttt{addr}(u,t) : (\Lambda_p \times \Lambda_p) \to \mathbb{N}$ utilisant une fonction $\texttt{addr}_i^r(u,t,k) : (\mathbb{N}\times\mathbb{N}\times\Lambda_{\texttt{p}}^{\texttt{aff}}\times\Lambda_{\texttt{p}}^{\texttt{aff}}\times\mathbb{N}) \to \mathbb{N}$:

$$\texttt{addr}(u,t) = \texttt{addr}_0^r(u,t,1) \qquad \texttt{addr}_i^r(t,t,k) = i \qquad \texttt{addr}_i(u,\lambda a.t,k) = addr_{i+k}(u,t,k)$$

$$\texttt{addr}_i^r(u,\langle t_0,...,t_{n-1}\rangle,k) = \texttt{addr}_{i+pos+k}(u,t_{pos},n\cdot k) \quad \text{si } u \in t_{pos}$$

$$\texttt{addr}_i^r(u,t_0[\langle\vec{x}\rangle := t_1],k) = \texttt{addr}_i^r(u,t_0t_1,k) = \begin{cases} \texttt{addr}_{i+k}^r(u,t_0,k) & \text{si } u \subseteq t_0 \\ \texttt{addr}_{i+r+2k-1}^r(u,t_1,k) & \text{si } u \subseteq t_1 \end{cases}$$

Le codage associe un entier de taille $O(log\ |t|)$ à chaque sommet de t vu comme un arbre syntaxique. L'adressage fonctionne par accumulation : on part de la racine avec une adresse initiale qu'on incrémente jusqu'à atteindre la position voulue.

La fonction utilise un paramètre r qui va faire perdre de la précision volontairement à l'adressage. Nous aurions pu avoir un codage exact et remplacer r par $|t_0|$ dans le cas de l'application t_0t_1 mais il semblerait que cela pose des difficultés techniques dont nous discuterons en conclusion.

Definition 40 (langage de connexion directe).

Soit $\mathcal{T} = (t_n)_{n\in\mathbb{N}}$ une famille de termes de $\Lambda_{\texttt{p}}^{\texttt{aff}}$. Son langage de connexion directe est donné par l'ensemble $L_{DC}(\mathcal{T})$ = l'ensemble des $(id, key, loc, type, code)$ où

— $id = \texttt{una}(n) \bullet 0^n$. Notamment on a $|id| = 2n$.

— $key = \texttt{bin}_{log(n)}(r)$ où

$$r \geq max\{|t_0| : \text{ il existe } t_1 \text{ tel que } t_0t_1 \text{ ou } t_0[\langle\vec{x}\rangle := t_1] \text{ est un sous terme de } t_{id}\}$$

— $loc = \texttt{bin}_{log(n)}(k)$ tel qu'il existe u satisfaisant $\texttt{addr}(u,t_{id}) = k$ (l'adresse d'un sous terme).

— $type = \texttt{bin}_{log(n)}(t)$ tel que :

— Si $t = 0$ alors $code \in \{\texttt{var}, \texttt{polyvar}, \texttt{app}, \texttt{abs}, \texttt{seq}, \texttt{let}, *\}$ (constantes).

— Si $t > 0$ alors $code = \texttt{bin}_{log(n)}(c)$ est l'adresse de la $type$-ième prémisse de t_{id}.

Definition 41 (uniformité d'une famille).

Pour une classe de complexité $\mathcal{C}$ donnée, une famille de termes de $\Lambda_{\texttt{p}}^{\texttt{aff}}$ définie par $\mathcal{T} = (t_n)_{n\in\mathbb{N}}$ est $\mathcal{C}$-uniforme si et seulement si $L_{DC}(\mathcal{T}) \in \mathcal{C}$.

Nous faisons le choix très strict d'utiliser une contrainte en temps logarithmique sur l'uniformité pour des raisons de standards, de généralité et car il semblerait que cela soit la plus petite classe valable pour les résultats d'uniformité.

Cependant, une machine de Turing en temps logarithmique est incapable de lire son entrée en entier. Nous définissons une variante des machines de Turing pouvant tourner en temps logarithmique venant de Barrington [BIS90].

Definition 42 (machine de Turing logtime).

Une machine de Turing logtime est une machine de Turing possédant :

— un ruban d'entrée de taille n en lecture seule

— un ruban d'adresse (pointant sur une case de l'entrée) de taille $log\ n$

— un nombre constant de rubans de travail de taille $O(log\ n)$

Definition 43 (potentiel des machines logtime).

Les machines de Turing logtime déterministes peuvent réaliser les opérations suivantes en temps $O(log\ n)$ pour une entrée de taille n :

— déterminer la taille de l'entrée (par recherche dichotomique)

— additionner et soustraire des entiers en binaire de $O(log\ n)$ bits

— calculer des opérations de modulo sur des entiers en binaire de $O(log\ n)$ bits

— déterminer le logarithme d'un entier en binaire de $O(log\ n)$ bits

Théorème 16 (DLOGTIME-uniformité d'approximations homogènes).

Pour tout $M \in \Lambda_{!}^{\texttt{aff}}$, la famille d'approximations homogènes $(\lfloor M \rfloor_k)_{k\in\mathbb{N}}$ est **DLOGTIME**-uniforme.

Preuve.

L'idée est que l'on a une entrée qui se réfère ou non à une position précise d'un terme $\lfloor M \rfloor_k$. Il faut vérifier si cette position a une correspondance dans M en parcourant son arbre syntaxique.

Montrons que $L_{DC}((\lfloor M \rfloor_k)_{k\in\mathbb{N}})$ est décidable en temps logarithmique par une machine de Turing logtime déterministe. Décrivons le fonctionnement d'une machine $\mathfrak{D}$ sur l'entrée $w \in \{0,1\}$ de taille $n = k + 4 \cdot log(k)$:

Localisation

On définit une procédure de localisation utilisant un n-uplet $(id, key, type, loc, code)$ pour localiser un terme référé par les composantes key, $type$, loc et $code$ du langage de connexion directe de $\lfloor M \rfloor_k$ pour un certain k. Nous supposons que si l'entrée n'est pas de la bonne forme alors la machine rejette et que si key n'est pas cohérent par rapport à la définition du langage de connexion directe alors le comportement de la machine n'est pas défini.

Le terme M étant constant (fixé), on peut supposer que l'arbre syntaxique de M est ancré dans le graphe d'états de $\mathfrak{D}$. On réalisera des opérations (en temps logarithmique) sur une adresse pour savoir comment parcourir l'arbre de M.

Les états de parcours sont étiquetés avec un élément de $\{\texttt{var}, \texttt{polyvar}, \texttt{app}, \texttt{abs}, \texttt{seq}, \texttt{let}, *\}$ représentant un constructeur de M.

1. On "dé-référence" une adresse i de la manière suivante :
   - Si on est sur un état $\texttt{var}, \texttt{polyvar}, \texttt{abs}$, $i := i - 1$, continuer de parcourir M
   - Si on est sur un état $\texttt{app}, \texttt{let}$ représentant respectivement les termes $t_0 t_1$ et $t_0[\langle \vec{x} \rangle := t_1]$:
     - Si $i > key$, alors $i := i - r$ et on continue dans t_1
     - Sinon, $i := i - 1$ et on continue dans t_0
   - Si on est sur un état $\texttt{seq}$, on inverse l'opération d'une manière similaire en utilisant des propriétés d'arithmétique modulaire.
2. On recommence l'étape précédente jusqu'à arriver à ces situations finales :
   - Si on est sur une feuille et $d \neq 0$, rejeter.
   - Sinon accepter.
3. On arrive sur une localisation dans M. Il reste à vérifier que cette localisation est conforme.
   - Si $type = 0$ et $code = *$, accepter.
   - Sinon si $type = 0$ vérifier si $code$ correspond au symbole de la position loc de M.
   - Sinon, on recommence une localisation avec $type := 0, loc := code, code := *$

Décision

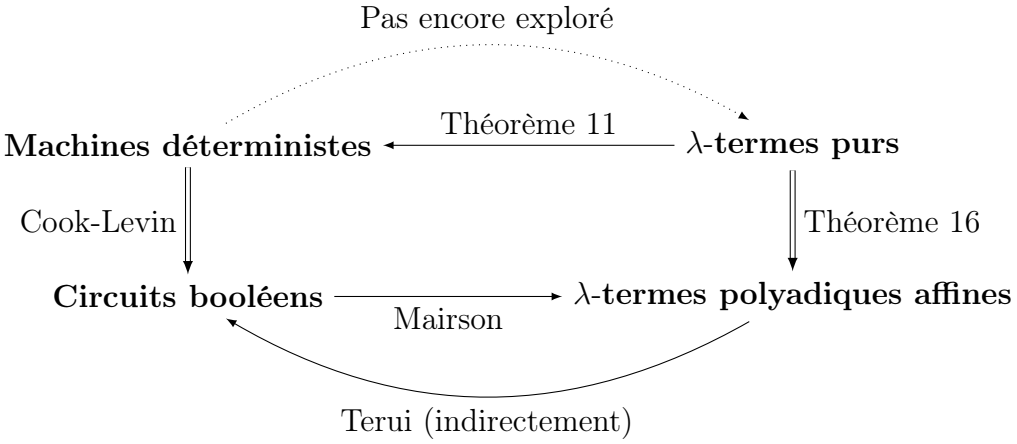
On accède au début de $type$ par recherche dichotomique O(log n) comme décrit dans [BIS90].

1. Si $type > 0$, on est sur l'élément d'une séquence. Il faut vérifier si la position est cohérente avec la taille des séquence k. Par recherche dichotomique on peut aller sur la position du milieu de id nous donnant $bin(k)$ sur le ruban d'adressage.
2. Si $type - bin(k) \neq 0$ alors $type > bin(k)$. Rejeter, car la taille des séquences polyadiques est limitée à k.
3. Pour terminer, on localise $type$, loc et $code$ dans M.

Complexité et terminaison
La décision et la terminaison sont évidents. De plus, on peut borner la complexité de la décision par un logarithme par composition des bornes de chaque opérations. Toutes les soustractions entre nombres binaires se font en temps $O(log\ n)$ par le lemme de potentiel des machines logtime.

□

Nous concluons avec le diagramme suivant déduit de nos deux théorèmes :



6 Conclusion

Dans ce document, nous avons exploré de nombreux outils provenant de la théorie du λ-calcul et de la logique linéaire et montré leur pertinence dans l'étude de la complexité. Nous avons pu donner une description de la simulation d'un λ-terme par une machine de Turing tout en gardant une décision en espace constant. De plus, nous avons montré l'uniformité de la famille de termes induite par ce dernier résultat afin de passer d'une vision mécanique à une vision algébrique du calcul.

6.1 Justification des choix

Cadre affine. Bien que plus strict et fort, nous n'avons pas privilégié le cadre du λ-calcul linéaire (les fonctions consomment exactement une fois leur argument) car il n'est pas adapté aux discussions sur l'approximation. Certains résultats comme la monotonicité de l'approximation polyadique par rapport à la réduction ne sont plus valides (la réduction connecte strictement les occurrences dans le cadre linéaire).

Modélisation des booléens. Il peut être intéressant de noter qu'il existe d'autres modélisations des booléens. Une utilisée par Terui [Ter04] est $Bool := (X \otimes X) \multimap (X \otimes X)$ qui a le mérite de marcher dans le cadre linéaire. Étant dans un cadre affine, cela nous pose aucun problème. Nous avons fait le choix le plus standard.

Modélisation des chaînes de caractères. Dans sa correspondance entre circuits booléens et réseaux de preuve de la logique linéaire, Terui [Ter04] montre qu'il est nécessaire de considérer des sous-formes potentiellement différentes pour les types. C'est pour cela que nous introduisons une *expansion de type* plutôt que d'utiliser de simples atomes X. Dans le cas contraire, Terui montre que certaines fonctions ne sont pas calculables alors qu'elles devraient l'être moralement.

Adressage approximatif. Nous aurions pu faire un codage exact (c'est la direction que nous avions prise au départ) mais cela pose des complications techniques. Dans le cas des termes $t_0 t_1$ il faut pouvoir calculer la taille de t_0. Étant restreint au temps logarithmique, cela pourrait nous poser problème pour décoder une adresse par une machine. Notre usage de *key* nous permet d'avoir une certitude en temps logarithmique dans la direction à prendre dans l'arbre syntaxique de M mais on perd en précision dans l'adressage.

Langage de connexion directe. N'utilisant pas de symboles de séparation, on utilise des chaînes binaires de même longueur organiser l'entrée. La première partie *id* contient une chaîne de bits inutiles afin d'utiliser une procédure de recherche dichotomique décrite dans [BIS90] et ainsi garder un temps logarithmique.

Restriction de l'uniformité. Il existe en effet une restriction d'uniformité introduite par Bonfante et Mogbil [BM12] qui est plus stricte que **DLOGTIME** mais nous n'avons pas pris le temps de l'explorer et avons finalement fait le choix d'utiliser la définition la plus simple et standard.

6.2 Ouverture possible

— Pour montrer qu'une bonne mesure de l'espace du λ-calcul donne un modèle de coût raisonnable, il reste à donner une simulation des machines de Turing en espace $f(n)$ par un λ-terme (éventuellement de $\Lambda_!^{\mathtt{aff}}$ mais pas nécessairement) typé par une dérivation en espace $O(f(n))$ dans un système de type intersection. Une première piste est d'explorer l'encodage des machines de Turing par un λ-terme proposé par Accattoli et Dal Lago [AL12].

6.3 Remerciements

- Je tiens avant tout à remercier **Michele Pagani** et **Delia Kesner** que je n'ai pas pu remercier lors de mon stage de L3 Informatique. Je considère ce court stage comme un bond important.
- Je remercie bien évidemment **Damiano Mazza** pour tout le temps qu'il m'a accordé dans le cadre de sa supervision pour mon travail de recherche encadré mais aussi pour son intérêt dans les questions que j'ai pu lui poser lors de mon stage de L3.
- Je remercie finalement **Clément Aubert** et **Luc Pellissier** pour avoir répondu avec intérêt (parfois passion) à mes questions par des échanges d'e-mails.

Références

[AK10] Beniamino Accattoli and Delia Kesner. The structural *lambda*-calculus. In *Computer Science Logic, 24th International Workshop, CSL 2010, 19th Annual Conference of the EACSL, Brno, Czech Republic, August 23-27, 2010. Proceedings*, pages 381–395, 2010.

[AL12] Beniamino Accattoli and Ugo Dal Lago. On the invariance of the unitary cost model for head reduction. In *23rd International Conference on Rewriting Techniques and Applications (RTA'12) , RTA 2012, May 28 - June 2, 2012, Nagoya, Japan*, pages 22–37, 2012.

[BIS90] David A Mix Barrington, Neil Immerman, and Howard Straubing. On uniformity within nc1. *Journal of Computer and System Sciences*, 41(3) :274–306, 1990.

[BM12] Guillaume Bonfante and Virgile Mogbil. A circuit uniformity sharper than dlogtime. 2012.

[Bor77] Allan Borodin. On relating time and space to size and depth. *SIAM journal on computing*, 6(4) :733–744, 1977.

[CD80] Mario Coppo and M Dezani. An extension of the basic functionality theory for the lambda-calculus. *Notre Dame journal of formal logic*, 21 :685–693, 1980.

[Chu32] Alonzo Church. A set of postulates for the foundation of logic. *Annals of mathematics*, pages 346–366, 1932.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[DC17] Daniel De Carvalho. Execution time of λ-terms via denotational semantics and intersection types. *Mathematical Structures in Computer Science*, pages 1–35, 2017.

[Gar94] Philippa Gardner. Discovering needed reductions using type theory. In *International Symposium on Theoretical Aspects of Computer Software*, pages 555–574. Springer, 1994.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical computer science*, 50(1) :1–101, 1987.

[Gir89] Jean-Yves Girard. Geometry of interaction 1 : Interpretation of system f. In *Studies in Logic and the Foundations of Mathematics*, volume 127, pages 221–260. Elsevier, 1989.

[How80] William A Howard. The formulae-as-types notion of construction. *To HB Curry : essays on combinatory logic, lambda calculus and formalism*, 44 :479–490, 1980.

[Mai04] Harry G Mairson. Functional pearl linear lambda calculus and ptime-completeness. *Journal of Functional Programming*, 14(6) :623–633, 2004.

[Maza] Damiano Mazza. Implicit computational complexity, non-uniformity and the lambda-calculus.

[Mazb] Damiano Mazza. On time and space in higher order boolean circuits.

[Maz12] Damiano Mazza. An infinitary affine lambda-calculus isomorphic to the full lambda-calculus. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012*, pages 471–480, 2012.

[Maz13] Damiano Mazza. Non-linearity as the metric completion of linearity. In *Typed Lambda Calculi and Applications, 11th International Conference, TLCA 2013, Eindhoven, The Netherlands, June 26-28, 2013. Proceedings*, pages 3–14, 2013.

[Maz16] Damiano Mazza. Church meets cook and levin. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, pages 827–836, 2016.

[Maz17a] Damiano Mazza. Infinitary affine proofs. *Mathematical Structures in Computer Science*, 27(5) :581–602, 2017.

[Maz17b] Damiano Mazza. Polyadic approximations in logic and computation. 2017.

[Mog01] Torben Æ Mogensen. An investigation of compact and efficient number representations in the pure lambda calculus. In *Perspectives of System Informatics : 4th International Andrei Ershov Memorial Conference, PSI 2001 Akademgorodok, Novosibirsk, Russia, July 2–6, 2001 Revised Papers 4*, pages 205–213. Springer, 2001.

[SBHG08] Daniel Spoonhower, Guy E Blelloch, Robert Harper, and Phillip B Gibbons. Space profiling for parallel functional programs. In *ACM Sigplan Notices*, volume 43, pages 253–264. ACM, 2008.

[Sch06] Ulrich Schöpp. Space-efficient computation by interaction. In *Computer Science Logic, 20th International Workshop, CSL 2006, 15th Annual Conference of the EACSL, Szeged, Hungary, September 25-29, 2006, Proceedings*, pages 606–621, 2006.

[Ter04] Kazushige Terui. Proof nets and boolean circuits. In *Logic in Computer Science, 2004. Proceedings of the 19th Annual IEEE Symposium on*, pages 182–191. IEEE, 2004.

[Tur37] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1) :230–265, 1937.

[vEB90] Peter van Emde Boas. Handbook of theoretical computer science (vol. a). chapter Machine Models and Simulations, pages 1–66. MIT Press, Cambridge, MA, USA, 1990.

[Vol99] Heribert Vollmer. Introduction to circuit complexity. texts in theoretical computer science, 1999.

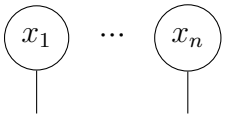
Annexes

Cette partie contient des informations qui ne sont pas strictement nécessaires à la lecture du document. On y inclut entre autre des preuves de lemmes et théorèmes auxiliaires et tout commentaire sur le contenu du document global

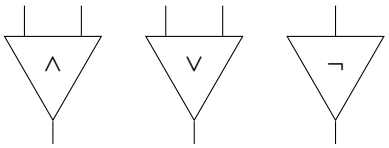
Annexe A - Circuits booléens

Definition 44 (circuits booléens).

Un circuit booléen C_n est un graphe orienté, acyclique et étiqueté contenant n entrées de la forme :



et formé de sommets :



La racine est appelée *sortie*. Un circuit représente une fonction booléenne

$$f : \{0,1\}^n \longrightarrow \{0,1\}$$

et peut aussi être vu comme une formule booléenne avec partage de sous-formules.

Definition 45 (évaluation).

L'évaluation booléenne $[\![C_n(w)]\!]$ d'un circuit C_n remplace tous les x_i par des valeurs de $\{0,1\}$ puis est définie inductivement :

- L'évaluation d'un sommet de label $b \in \{0,1\}$ et le booléen correspondant
- L'évaluation d'un sommet n-aire de prémisses $c_1, ..., c_n$ est de label l est $f([\![c_1]\!], ..., [\![c_n]\!])$ où f est la fonction booléenne associée à l (par exemple le label $\wedge$ correspond à $\texttt{and} : \{0,1\} \times \{0,1\} \longrightarrow \{0,1\}$ réalisant la conjonction).

L'évaluation du circuit entier est l'évaluation de sa racine.

Definition 46 (acceptation des circuits).

Un circuit C_n accepte un mot $w = a_1 \bullet \cdots \bullet a_n \in \{0,1\}^n$ si et seulement si l'évaluation booléenne du circuit où les entrées $x_1, ..., x_n$ sont remplacées par $a_1, ..., a_n$ donne 1, ce qu'on note $[\![C_1(w)]\!] = 1$. De plus, si le circuit accepte tous les mots d'un langage $\mathcal{L} \in \{0,1\}^n$ alors il accepte ce langage.

Annexe B - Directions parallèles

- Le calcul polyadique de Mazza peut en fait être vu comme une version déterministe du λ-calcul avec de ressources introduit par Boudol.

— Le travail de Mazza n'est pas la seule interprétation du théorème d'approximation de Girard. Par exemple, Melliès, Tabareau et Tassons en donnent une interprétation catégorique.

— Il existe plusieurs variantes de λ-calcul infinitaires dont certaines fondées sur une notion de réécriture infinitaire. La littérature est introduite par Dershowitz et al. En particulier Kennaway et al. propose un calcul infini en profondeur (qui s'oppose à celui de Mazza infini en largeur).

— Des travaux comme ceux d'Aubert, Mogbil et de Terui rapprochent les circuits booléens des réseaux de preuves de la logique linéaire. Les liens entre logique linéaire et λ-calcul étant plutôt connus et développés nous nous sommes concentré sur le lien entre circuits et termes.

Références.

Boudol, G. (1993, August). The lambda-calculus with multiplicities. In International Conference on Concurrency Theory (pp. 1-6). Springer, Berlin, Heidelberg.

Melliès, P. A., Tabareau, N., & Tasson, C. (2017). An explicit formula for the free exponential modality of linear logic. Mathematical Structures in Computer Science, 1-34.

Dershowitz, N., Kaplan, S., & Plaisted, D. A. (1991). Rewrite, rewrite, rewrite, rewrite, rewrite,.... Theoretical Computer Science, 83(1), 71-96.

Kennaway, J. R., Klop, J. W., Sleep, M. R., & de Vries, F. J. (1997). Infinitary lambda calculus. Theoretical Computer Science, 175(1), 93-125.

Terui, K. (2004, July). Proof nets and boolean circuits. In Logic in Computer Science, 2004. Proceedings of the 19th Annual IEEE Symposium on (pp. 182-191). IEEE.

Mogbil, V., & Rahli, V. (2007, June). Uniform circuits, & Boolean proof nets. In International Symposium on Logical Foundations of Computer Science (pp. 401-421). Springer, Berlin, Heidelberg.

Aubert, C. (2012). Sublogarithmic uniform boolean proof nets. arXiv preprint arXiv :1201.1120.

Annexe C - Prise de recul sur la complexité

Complexité classique. Les premiers pas de la complexité traditionnelle ont commencé avec les machine de Turing. Très rapidement, les pionniers sont tombés sur des problèmes inabordables comme $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ et se sont rendu compte de l'inefficacité de leurs méthodes. Les problèmes de *preuve naturelle*, *l'algébrisation* et de *relativisation* posent un premier obstacle dans le monde de la complexité. Il semblerait que notre vision du calcul était encore trop opaque à ce moment. Des tentatives "d'ouvrir" le calcul ont été faites à travers les circuits booléens entre autres. Ces directions ont mené vers de plus grandes complications débouchant sur des problèmes de circuits arithmétiques.

Complexité implicite et descriptive. Plus tard, plusieurs mouvements provenant principalement de la logique et de l'informatique proposent une nouvelle vision de la complexité. L'idée est de caractériser les classes de complexité par un langage de programmation ou un système logique. Une classe de complexité est vu comme un ensemble de contraintes sur un langage formel. Ainsi, la décision dans le langage qui caractérise **P** sera forcément en temps polynomial par construction. Cette branche a été largement développée par la logique linéaire à travers les logiques légères. Est-ce une voie intéressante pour franchir les murs de la complexité ? Est-ce que cette voie a ses limites ? Lesquelles ?

Complexité géométrique. Une nouvelle branche parallèle de la complexité appelée *théorie géométrique de la complexité* propose d'utiliser des méthodes mathématiques de *géométrie algébrique* et de *théorie de la représentation*. On retrouve quelques investigations toujours liées à la logique linéaire dans les travaux de Thomas Seiller.

Conclusion. Il semblerait qu'à l'heure actuelle, au niveau conceptuel, l'organisation de la complexité est encore bien confuse et que la nature de la logique et du calcul ne sont toujours pas à notre portée. Qu'est-ce qu'est réellement un programme ? La réponse actuelle que nous avons semble être biaisée par notre vision naïve du calcul qui visiblement peut prendre des formes très disparates.

Notre travail se place dans une approche mixte où l'on utilise des outils de la complexité implicite dans un cadre de complexité traditionnelle. Il semblerait qu'il y ait des premières tentatives d'unifier certaines concepts de la complexité à travers des constructions catégoriques (Mazza).

Annexe D - Lemmes auxiliaires

Lemme 17 (typage des chaînes binaires).

Pour tout $w \in \{0,1\}^n, \vdash \underline{w} : Str_n[\overrightarrow{B^n}]$.

Preuve.

Par cas sur la longueur de w :

— $w = \varepsilon$. On a la dérivation suivante avec un contexte Γ dont on ne se préoccupe pas

$$\dfrac{\dfrac{}{\Gamma, e : B_0^0 \vdash e : B_0^0}}{\vdash \lambda s_0.\lambda s_1.\lambda e.e : Str_0[\overrightarrow{B^0}]}\,\lambda^*$$

— $w = w_0 \cdot (w_1 ... w_n)$. Montrons que $\vdash \underline{w} : Str_{m+1}$. On applique autant de règle d'abstraction que possible pour obtenir

$$\dfrac{\dfrac{\vdots}{e : B_0^n, s_0 : \langle - \rangle^{n+1}, s_1 : \langle - \rangle^{n+1} \vdash s_{w_0}(s_{w_1}...(s_{w_n}\ e)...) : B_n^n}}{\vdash \lambda s_0.\lambda s_1.\lambda e.s_{w_0}(s_{w_1}...(s_{w_n}\ e)...) : Str_{n+1}[\overrightarrow{B^{n+1}}]}\ \lambda^*$$

On omet d'écrire les types en entier en mettant $\langle - \rangle^n$ pour la séquence de type attendue avec une taille de n. Il s'agit ensuite de détacher chaque s_{w_0} un par un et de les connecter à une ressource de s_0 ou s_1 que l'on consomme par la règle d'application. Si on prend, sans perdre de généralité, $w_0 = 0$ alors on a

$$\dfrac{\dfrac{\dfrac{}{s_0 : B_i \to B_{i+1} \vdash s_0 : B_i \to B_{i+1}} \quad \vdots}{e : B_0^n, s_0 : \langle - \rangle^{n+1}, s_1 : \langle - \rangle^{n+1} \vdash s_0(s_{w_1}...(s_{w_n}\ e)...) : B_n^n}}{\vdash \lambda s_0.\lambda s_1.\lambda e.s_0(s_{w_1}...(s_{w_n}\ e)...) : Str_{n+1}[\overrightarrow{B^{n+1}}]}\ \lambda^*$$

Dans la prémisse de droite, on continue avec la suite de la chaîne binaire et le reste des ressources :

$$\dfrac{\vdots}{e : B_0^n, s_0 : \langle - \rangle^{m}, s_1 : \langle - \rangle^{m+1} \vdash (s^1_{w_1}...(s^n_{w_n}\ e)...) : B_n^n}{\vdots}$$

En répétant la procédure jusqu'au bout on obtient une application de règle axiome où une partie de s_0 et s_1 ont été consommés (ce qui ne nous gêne pas grâce à l'affinité) :

$$\dfrac{}{e : B_0^n, s_0 : \langle - \rangle^{i}, s_1 : \langle - \rangle^{j} \vdash e : B_n^0}$$
$$\vdots$$

□

Lemme 18 (négligence spatiale).

Pour tout M constant (fixé),

$$|lin(\lfloor M \rfloor_{\kappa(n)})| \leq |M|^2 \cdot (\kappa(n) + 1)^{2 \cdot expdepth(M)}$$

c'est-à-dire que la taille de la linéarisation de l'approximation cartésienne homogène est négligeable par rapport au polynôme $|M|^2 \cdot (\kappa(n) + 1)^{2 \cdot expdepth(M)}$.

Preuve.

On commence par montrer $|\lfloor M \rfloor_{\kappa(n)}| \leq |M| \cdot (\kappa(n) + 1)^{expdepth(M)}$ par induction sur M.

— $M = a$. On a $|a| = 1$ qui est bien borné par $|a| \cdot (\kappa(n) + 1)^0 = 1$

— Idem pour $M = x$.

— $M = \lambda a.M'$. L'hypothèse d'induction est $|\lfloor M' \rfloor_{\kappa(n)}| \leq |M| \cdot (\kappa(n)+1)^{expdepth(M')}$.
On a démarre avec

$$|\lfloor \lambda a.M' \rfloor_{\kappa(n)}| = 1 + |\lfloor M' \rfloor_{\kappa(n)}|$$

qu'on peut borner, par hypothèse d'induction, par

$$|M'| \cdot (\kappa(n)+1)^{expdepth(M')}$$

lui-même borné par

$$(1+|M'|) \cdot (\kappa(n)+1)^{expdepth(M')} = |\lambda a.M'| \cdot (\kappa(n)+1)^{expdepth(\lambda a.M')}$$

— Preuve similaire pour $M = !M'$.

— $M = (M_1 M_2)$. On a deux hypothèses d'induction :
$|\lfloor M_1 \rfloor_{\kappa(n)}| \leq |M_1| \cdot (\kappa(n)+1)^{expdepth(M_1)}$ et $|\lfloor M_2 \rfloor_{\kappa(n)}| \leq |M_2| \cdot (\kappa(n)+1)^{expdepth(M_2)}$
On démarre avec

$$|\lfloor M_1 M_2 \rfloor_{\kappa(n)}| = |\lfloor M_1 \rfloor_{\kappa(n)}| + n \cdot |\lfloor M_2 \rfloor_{\kappa(n)}|$$

qu'on peut borner, par hypothèses d'induction, par

$$\begin{aligned}
&|M_1| \cdot (\kappa(n)+1)^{expdepth(M_1)} + \kappa(n) \cdot |M_2| \cdot (\kappa(n)+1)^{expdepth(M_2)} \\
&\leq (|M_1| + |M_2|) \cdot (\kappa(n)+1)^{max(expdepth(M_1), expdepth(M_2))} \\
&\leq (|M_1| + |M_2| + 1) \cdot (\kappa(n)+1)^{max(expdepth(M_1), expdepth(M_2))} \\
&= |M_1 M_2| \cdot (\kappa(n)+1)^{expdepth(M_1 M_2)}
\end{aligned}$$

— Preuve similaire pour $M = M_1[!x \coloneqq M_2]$.

La linéarisation de $\lfloor M \rfloor_{\kappa(n)}$ fait croître le terme d'au plus $|M|$ ce qui borne son espace par $|M| \cdot (\kappa(n)+1)^{expdepth(M)} \cdot |M| \cdot (\kappa(n)+1)^{expdepth(M)} = |M|^2 \cdot (\kappa(n)+1)^{2p(M)}$.

□