Parcours et Projet d'intégration

Rapport Analytique

Auteur
Valentin Maestracci

Candidature MCF
Année universitaire : 2026

1 : Identité et Parcours universitaire

1.1 - Contact

- **Nom:** Maestracci Valentin
- **Adresse:** 1270 route du canal de Provence, 83210 Belgentier
- **Email:** valentin.maestracci.academic@proton.me
- **Site web:** https://vmaestracci.github.io/

1.2 - Parcours Universitaire

- L3 ENS Cachan (2017-2018)
- M1 MPRI[1] (ENS Cachan[2]) (2018-2019)
- M2 MPRI[1] (ENS Cachan[2]) (2019-2020)
- Césure: Ecole de Dessin (2020-2021)
- ARPE: Année recherche pré-doctorale (2021-2022)
- Doctorat en Mathématique[3] (I2M - Université d'Aix Marseille): Réalisabilité pour la Logique Linéaire (2022-2025)
- Post-Doc (IRISA - Université de Rennes): Sémantique des jeux pour améliorer l'assistant de preuve cryptographique Squirrel

1.3 - Publications et Talk

Publication actuellement publiée:

- Functorial Models of Linear Logic (MFPS 202) 5[1]
- The Lambda Calculus is Quantifiable (CSL 2025) [2]
- Implémentation de 2LTT dans Dedukti (LMFPT 2020)[3]

Publications en cours:

- From Geometry to Interaction (soumise en 2026)
- Multiplicative linear logic from a resolution-based tile system (soumise en Journal, en révision)

[1] Master Parisien de Recherche en Informatique

[2] Actuellement ENS Paris-Saclay

[3] Mon doctorat est en Mathématique car réalisé dans un laboratoire de Mathématique, mais au sein d'une équipe de Logique et Calcul, le contenu est très Informatique

- A purely local account of additives (Titre provisoire, en écriture)

Présentations:

- Séminaires et Workshop:
  - Séminaire LACL 2026: « How Logic describes the behaviours of Programs: Introduction to Linear Realisability through Interaction Graphs »
  - Realizability Workshop 2025 Linear Realisability and Types
  - Chocola Invited Talk 2025 on my paper: « The lambda calculus can be Quantified » [2]
  - Séminaire Doctorant (LIPN/I2M) 2024: Don't take Math for granted!
- Workshop Talk:
  - TLLA 2025: « Drifter;Graphs : a dynamically-sliced model for Additives »
  - GALOP 2025: « Linear Realisability and Cobordism: Understanding the Trefoil Property »
  - TLLA 2023: « Linear Realisability and Cobordisms »
  - TLLA 2023, GT SCALP 2023: « Functorial Models of Differential Linear Logic »
- Vulgarisation:
  - Présentation Pi-day 2023 (AMU): Les apparitions innatendues de π

1.4 - Autres expériences

Stages de Recherche:
- L3 (2 mois): Implémenter les complexe de chaine en Cubical Agda
- M1 (6 mois): Etude de programme PV via la Topologie Algébrique Dirigée
- M2 (4 mois): Définir les 2LTT dans Dedukti

Ecoles de recherche:

- CIRM, Ecole de Printemps d'Informatique Théorique (EPIT), 2026: Probabilistic Programming
- CIRM, 2025: Advances in Interactive and Quantitative Semantics
- CIRM, 2O25: Complexity as a kaleidoscope
- Midland Graduate School (MGS) 2024
- CIRM, 2024: Differential Lambda-Calculus and Differential Linear Logic, 20 Years Later
- Midland Graduate School (MGS) 2023
- Summer School on Foundations of Programming and Software Systems (FOPS) Bertinoro, 2023: Quantitative Aspects of Program Semantics, Verification, and Transformation
- CIRM, 2023: Type Theory, Constructive Mathematics and Geometric Logic
- CIRM, 2022, Logic and higher structures 21-25 February
- CIRM, 2022: Logic and transdiciplinarity: Mathematics/Computer Science/ Philosophy/Linguistics
- CIRM, 2022, Linear Logic Winter School

1.5 - Diplômes

- Bac S option Mathématiques (Lycée du Coudon, 2015) : Mention Très Bien.
- L3 Informatique ENS Cachan[2] (2018): Mention TB
- M2 MPRI[1] ENS Cachan[2] / Paris 7 (2020): Magna Cum Laude

- Doctorat en Mathématique de l'université d'Aix-Marseille
- Cambdridge Advanced niveau C2

2 : Résumés des Travaux

Ma recherche se trouve au coeur de la correspondance preuves-programme (aussi appellée correspondance de Curry-Howard). J'étudie donc les interactions entre la logique (propositions et preuves) et les modèles de calcul / langages de programmation (types et programme), à des niveaux d'abstraction différent.

Pour exemple, voilà deux règles qui apparaissent, l'une dans un système de preuve en logique (Déduction Naturelle) et l'autre dans le système de typage d'un modèle de calcul minimaliste (le λ-calcul simplement typé):

$$\frac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B} \to_e$$

Fig. 1. – Elimination de l'implication (Modus Ponens)

$$\frac{\Gamma \vdash f : A \to B \quad \Gamma \vdash a : A}{\Gamma \vdash f(a) : B} \to_e$$

Fig. 2. – Application de fonction, λST

On remarque que ces deux règles sont identiques, au détail près que celle de droite fait apparaitre le corps du programme (le terme $f(a)$). Il y a donc une véritable interaction entre logique et programme. Notamment:

- La logique peut-être utilisée comme système de typage/spécification, donc pour forcer des programmes à adopter certains comportements. On peut notamment faire de la complexité implicite, ce qui permet de définir des attaquants polynomiaux dans ma recherche de Post-Doc (Chapitre 3). Plus généralement, elle permet de décrire ces comportements: c'est le coeur de la Réalisabilité et de mes travaux de thèse (Chapitre 2.3), on peut d'ailleurs se poser la question face à un nouveau comportement de quels règles logiques pourraient le capturer ? Cela permet de donner une justification aux règles logique, plus primitive que notre simple intuition.
- Les programmes peuvent permettre de mieux comprendre les propositions et axiomes mathématiques. Si l'on pense à l'énoncé d'un théorème comme une spécification, une preuve est un programme vérifiant cette spécification. On peut se demander quel genre de comportement est induit par ces spécification: par exemple, le tiers exclus est équivalent à l'élimination de la double négation: $\neg\neg A \Rightarrow A$. En terme calculatoire, cela reviens à se donner dans son langage de programmation un système d'exceptions.

Il existe une énorme variété de modèles de calculs (automates, machine de turing, λ-calcul, tuiles de wang, prolog etc…), mais parmis ceux ci, le λ-calcul se démarque de par l'immédiateté de ses liens avec la logique. Il est de plus notoirement « minimaliste » et au coeur de toute la programmation fonctionelle. L'étude du λ-calcul est donc un très bon départ pour mieux comprendre cette famille de langage de programmation, et c'est ce qui m'a poussé à l'étudier (Chapitre 2.2)

Il existe aussi une énorme variété de logique (premier ordre, ordre supérieur, type dépendant, modales, temporelles, substructurelles etc…). Parmis celles ci, la Logique Linéaire s'est distinguée car elle à permis de mieux comprendre de nombreux phénomènes calculatoires, menant par exemple à l'introduction du L-calcul polarisé [4], qui est un langage qui permet de beaucoup mieux manipuler et comprendre le flôt de contrôle des programmes fonctionnels (les exceptions sont notamment des manières de changer ce flot de contrôle). J'ai donc décidé d'étudier dans ma thèse la Logique Linéaire

sous l'angle des modèles de calcul (Chapitre 2.3). J'avais avant déjà décidé d'étudier une extension assez récente de LL, la Logique Linéaire Différentielle (Chapitre 2.1).

Mes travaux s'insèrent tous au sein de cette correspondance: j'aime étudier les modèles de calculs d'une part, la logique d'autre part, et quand il y en à les interactions entre les deux.

2.1 - Catégorie et Logique Linéaire Différentielle

C'est après une année de césure (celle du Covid), que j'ai commencé à choisir ma direction de recherche. J'avais été très intéressé en Master par les cours de Logique Linéaire, et j'ai été mis en contact pour mon année de recherche pré-doctorale (4A ENS) avec Marie Kerjean pour étudier la Logique Linéaire Différentielle, ce qui a donné lieu à une publication: Functorial Models of Differential Linear Logic [1].

Contexte Scientifique

La Logique Linéaire (LL) est une logique de ressources. En logique classique, on a que $A \Rightarrow A \land A$. Si on interpête A comme une connaissance, ce n'est pas un problème. Mais si A représente une ressource, on viens de la dupliquer, ce qui n'est à priori pas possible. Les opérateurs habituels, tels que « et » ($\land$) sont raffinés en des opérateurs plus précis, un et dit « additif », & et un « et » dit multiplcatif, $\otimes$

Pour illustrer la différence, voilà des descriptions de boulangerie en logique linéaire:

- 5$ ⊸ Pain ⊗ 5$, cette boulangerie n'est pas très réaliste: pour 5 dollar, on peut acheter du pain et obtenir 5$, donc on a rien perdu.
- 5$ ⊸ Pain &5$, cette boulangerie l'est déjà plus: pour 5 dollar, on peut choisir entre obtenir du pain et garder nos 5 dollars.

Cependant, aucune de ces deux boulangerie n'est réaliste sur un point: l'utilisation de $\multimap$ au lieu de $\Rightarrow$ veut dire qu'on ne peut y effectuer qu'un seul achat ! Notez aussi que les deux phrase utilisent bien le mot « et » mais que les sens sont subtilement différent, à cause de la présence d'un choix dans la seconde.

L'opérateur le plus important de LL est cet opérateur $\multimap$, l'implication linéaire, qui correspond à une fonction qui utilise linéairement ses arguments: $A \multimap A \otimes A$ n'est pas possible. On peut ensuite retrouver l'implication habituelle grace à un opérateur !, qui veut dire intuitivement « je peut dupliquer cet objet »: $A \Rightarrow B \equiv\ !A \multimap B$.

La Logique Linéaire Différentielle est une extension de la Logique Linéaire. Historiquement, la Logique Linéaire a été tirée de modèles ou l'un de ses opérateurs principaux, l'implication linéaire $\multimap$, correspondait a des fonctions linéaires. On s'est ensuite rendu compte que dans certains de ces modèles, il y avait des fonctions $\mathcal{C}^\infty$ qui correspondaient aux fonctions « usuelles » ($\Rightarrow$). La différentielle d'une fonction $\mathcal{C}^\infty$ est la meilleure approximation linéaire qu'on puisse en faire, et ce même lien existait entre $\Rightarrow$ et $\multimap$ dans ces modèles. On a pu ainsi en tirer une logique dans laquelle il est possible de dériver des preuves !

Le coeur de la Logique Linéaire peut-être décrit catégoriquement par des transformations naturelles correspondant aux opérateurs w, c, d, p à gauche dans la table suivante:

Opérateur	Type	Dual	Type	Intuition
w	$!A \multimap 1$	$\overline{\mathrm{w}}$	$1 \multimap !A$	Evaluation en 0
c	$!A \multimap !A \otimes !A$	$\overline{\mathrm{c}}$	$!A \otimes !A \multimap !A$	Convolution
d	$!A \multimap A$	$\overline{\mathrm{d}}$	$A \multimap !A$	Différentiation
p	$!A \multimap !!A$	-	-	-

Tableau 1. – Dualité LL/DiLL

La Logique Linéaire Différentielle introduit quant à elle des opérateurs duaux $\overline{w}, \overline{c}, \overline{d}$. On requiert ensuite plusieurs axiomes sous la forme de diagrammes dont on demande la commutation, qui encodent le contenu calculatoire des preuves, via des interactions entre transformations naturelles (typiquement $\overline{d}$ correspond à une différentiation, et donc doit respecter la règle de la chaine).

Cependant, cette présentation de LL en catégorie n'est pas la présentation standard: elle est trop proche de la syntaxe (chaque opérateur correspon à une règle de la logique), et pas assez des modèles.

Le consensus dans la communautée de la Logique Linéaire est que la bonne notion de modèle abstraite (catégorique) pour la Logique Linéaire Multiplicative (MLL) est celle de Catégorie $*$-autonome, $\mathcal{L}$ dans Fig. 3.

Pour obtenir les exponentielles et ainsi récuperer la « vraie » Logique Linéaire (LL), il faut ensuite obtenir une « adjonction linéaire-non linéaire » entre une telle catégorie et une Catégorie Cartésienne Close (CCC), $\mathcal{C}$ dans Fig. 3.

$$(\mathcal{C}, \times) \underset{U \approx d}{\overset{\mathcal{E}' \approx p}{\rightleftarrows}} (\mathcal{L}, \otimes, \diamond)$$

Fig. 3. – Adjonction Linéaire-non Linéaire

Cette présentation est fonctorielle: on a un foncteur d'oubli U qui ne garde que la structure de CCC, et un foncteur $\mathcal{E}'$ tel que l'opérateur exponentiel $! \coloneqq \mathcal{E}' \circ U$.

Notez que c'est équivalent à la présentation en terme d'opérateurs: le foncteur U contient intuitivement la même information que la transformation naturelle d dite de « déréliction », et $\mathcal{E}'$ celle de p dite de « promotion » (ce sont des règles d'inférence de LL). Cependant, cette présentation fonctorielle à l'avantage d'être plus concise, plus proche des modèles réels (ce qui permet de prouver plus facilement qu'une catégorie concrète donnée est bien une instance de cette construction), et possède moins de diagrammes commutatifs dans son axiomatisation, donc moins de choses à prouver pour vérifier qu'on a bien affaire à un modèle.

La problématique

Lorsque j'ai commencé mes travaux sur le sujet, la présentation « canonique » de DiLL en catégorie était insatisfaisante: on demandait à obtenir une adjonction comme dans la Fig. 3, plus les opérateurs duaux de « weakening », de « contraction » et de « déréliction » dans Tableau 1, ce qui casse la symétrie: on a un modèle hybride avec des foncteurs et des transformations naturelles.

Notre travail à définit une axiomatique catégorique qui se veut équivalente (sous certaines hypothèses) mais fonctorielle en introduisant un foncteur $\vec{D}$ qui encode la transformation $\overline{d}$.

- La propriété de fonctorialité est la suivante: pour $f : A \to B, g : B \to C$, on a, en notant « ; » la composition séquentielle: $F(f; g) = F(f); F(g)$
- La règle de la chaine s'énonce ainsi, pour $f : A \to B, g : B \to C$ on a: $D_a(f; g) = D_a(f); D_{f(a)}(g)$

$$D_a(g \circ f) = D_{f(a)}(g) \circ D_a(f)$$
$$D_a(g \circ f) = D_{f(a)}(g : B \to C) \circ D_a(f : A \to B)$$

On remarque que la règle de la chaine est presque fonctorielle mais pas exactement: on a un indice $f(a)$ qui apparait.

L'idée est de transformer les fonction $f : A \to B$ en fonctions « pointée »: $f_a : (a, A) \to (b, B)$ avec $b = f(a)$. On prend alors comme foncteur $\vec{D}(f_a) = D_a(f)$. On a bien que $\vec{D}(f_a; g_b) = D_a(f; g) = D_a(f); D_{f(a)}(g) = \vec{D}(f_a); \vec{D}(g_b)$.

On peut construire une catégorie $I \downarrow \mathcal{C}$ contenant ces fonctions, et réexprimer toutes les propriétés attendues dans ce setting (sous quelques hypothèses). On obtient la présentation catégorique suivante:

$$\begin{array}{ccc} I \downarrow U & \xrightarrow{\vec{U}} & I \downarrow \mathcal{C} \\ \Big\downarrow \Pi & \swarrow_{\vec{D}} & \Big\downarrow \Pi \\ (\mathcal{L}, \diamond) & \xrightarrow{U} & (\mathcal{C}, \times) \end{array}$$

Cela permet d'avoir les avantages d'une présentation purement fonctorielle pour DiLL et non plus seulement pour LL. La règle de la chaine n'est plus exprimée axiomatiquement comme un diagramme mais devient naturellement la fonctorialité de $\vec{D}$. Ceci met en évidence aussi la dualité entre U et $\vec{D}$ qui est la version fonctorielle de celle $d/\overline{d}$.

On a en fait généralisé un peu plus: nos catégories possèdent une notion de différentiation, mais on a aussi exprimer des lien avec les catégories possédant une notion d'intégrale, sous une forme d'adjonction.

2.2 - Lambda-Calcul et Métriques

J'ai voulu, en parallèle de l'aspect logique, étudier dans ma recherche des aspects plus proche de la programmation. J'ai donc effectué la deuxième partie de mon année de pré-recherche sous la direction de Paolo Pistone. Cela a donné lieu à une publication: The Lambda Calculus Is Quantifiable [2]

Contexte Scientifique

Nous cherchions à étudier des métriques sur les programmes. Traditionellement, les spécialistes des langages de programmation s'intéressent plutôt à des équivalences, typiquement en λ-calcul, l'équivalence observationelle. Un contexte est un programme avec un trou, qui va prendre u ou v en argument et observer son comportement. Deux programmes u et v sont alors observationellement équivalents si pour tout contexte $C[]$, $C[u]$ termine ssi $C[v]$ termine. (La terminaison est la seul observation que l'on peut faire en λ-calcul pur, mais l'on pourrait par exemple demander en présence d'entier dans le langage de programmation, que $C[u]$ et $C[v]$ se réduisent en le même entier).

L'intêret d'utiliser des métriques plutôt que des équivalences est que cela permet en théorie d'obtenir des informations plus fine. Un exemple simple est à trouver du coté du machine learning: si l'on prend deux classifieurs chien/chat, c'est à dire deux programme p et q, qu'on a entrainés, et qui prennent des photos en entrée (sous la forme de tableau de pixel) en renvoyant 0 pour dire que c'est un chat, 1 pour dire que c'est un chien. Ce problème n'a pas de solution formelle, que se passe t'il si je donne une photo de cheval a p et q ? On entraine des IA à le résoudre du mieux que l'on peut, mais il n'y a aucun espoir pour que p et q donne le même résultat sur toute entrée ! On peut tout au plus espérer dire que ces deux programmes sont « proches » s'ils s'accordent sur beaucoup d'entrée.

Nous avons repris un papier de Paolo [5] pour essayer d'étudier de telles métriques. En pratique, une notion naturelle de métrique apparue dans ce contexte est celle de métrique partielle: ce sont presque des métriques, à la différence qu'il est possible que $d(x, x) \neq 0$.

Un bon exemple à avoir en tête pour ces outils est celui des « tailles d'intervalles ». Une information partielle sur un nombre peut être modélisé par un intervalle, qui donne une incertitude sur quelle est la « vraie » valeur du nombre: par exemple, l'information partielle $0.XXX...$ qui est un nombre dont connait seulement la première décimale, peut être représenté par l'intervalle $I = [0, 1]$. Similairement pour $2.XXX...$ et $J = [2, 3]$, comme illustré sur le schéma suivant:

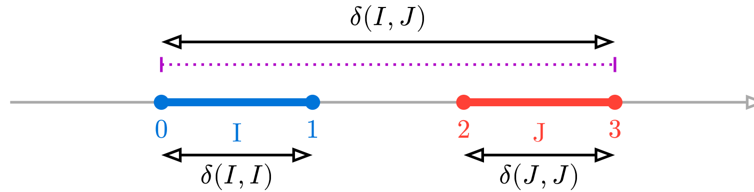


Fig. 4. – I, J, et la distance entre les deux

On peut alors définir une métrique partielle sur ces intervalles I et J: la distance entre deux nombres partiels est la taille $\delta(I, j)$ du plus petit intervalle qui contient les deux (en violet ici).

L'intêret est de prendre en compte le pire cas, ou les vraies valeurs des nombres seraient sur les bords « opposés » des deux intervalles.

La métrique partielle $\delta(-, -)$ n'est pas une métrique au sens usuel, car ici, $\delta(I, I) = 1 \neq 0$.

La problématique

Lors de l'étude de cette métrique partielle, nous nous sommes rendu compte que sa topologie induite est en fait une topologie bien connue en lambda-calcul: la topologie de Scott.

En sémantique du lambda calcul, et pour modéliser les programmes fonctionnels, c'est un outil classique de théorie des domaine. Les programmes sont alors interprétés comme des fonctions continues sur cette topologie. Intuitivement, cette topologie représente le fait qu'un programme calcule une information finie à partir d'une donnée finie. Par exemple, si on a 2 bit d'entrée « xx », découvrir la valeur d'un bit, disons le premier « 1x » permet d'approcher la valeur de sortie de la fonction. (Notez que l'on utilise des informations partielles).

Il est communément admis que cette Topologie est « compliquée ». En effet, elle n'est pas métrisable, les intuitions classiques venues des espaces métriques sont donc faussées.

Lors de nos travaux, nous avons en fait redécouvert un vieux programme de recherche qui montre que, bien que cette topologie ne soit pas métrisable, elle l'est « presque »: on peut en effet capturer cette topologie en utilisant la notion de « métrique partielle », comme dans l'exemple mentionné plus haut !

Nos contributions ont ensuite été d'utiliser ces métriques partielles pour caractériser des propriétés du Lambda-Calcul qui sont traditionellement exprimée topologiquement, mais dans un cadre (partiel)-métrique, ce qui permet des raisonnements plus intuitif: les raisonnements communément appelés « ε/δ » ! Nous avons aussi essayer d'insister sur le fait qu'il faut se méfier des idées reçues que l'on a sur un sujet !

On arrive ainsi à:

- Caractériser l'équivalence contextuelle, et la λ-théorie des arbres de Böhm en termes d'équivalence pour des métriques-partielles fixées. On arrive donc à recaractériser l'équivalence contextuelle, l'objet d'intéret habituel des spécialistes, mais dans le cadre (partiel)-métrique ce qui fait le lien entre les travaux anciens et plus modernes.
- Obtenir des modèles au sens catégorique, du Lambda-Calcul simplement typé et non-typé (via un objet réflexif et la construction D^∞ de Scott) qui utilisent des espaces partiel-métriques plutot que des espaces topologiques. On peut donc faire des raisonnements ε/δ sur ces programmes.
- Caractériser des informations sur le dévellopement de Taylor d'un lambda terme (notion qui est historiquement très liée à DiLL, Chapitre 2.1), via certaines métriques.

2.3 - Modèles de Calculs pour la Réalisabilité Linéaire (Sujet de Thèse)

Après avoir effectuée mon année ARPE, ou j'ai donc étudié d'une part un modèle de calcul, et d'autre part une logique, j'ai décidé de me lancer à l'intersection des deux. J'ai donc démarré ma thèse sous la direction de **Thomas Seiller** (CNRS, LIPN) et **Laurent Regnier** (Prof. HDR, Aix-Marseille Université).

Il est à noter que mes deux travaux mentionnés précédemment (Chapitre 2.1, Chapitre 2.2) étaient encore en cours même si les idées principales avaient été dévellopées, il y avait quelques blocages techniques que nous avons résolus pendant ma thèse, d'ou leur date de sortie.

Contexte Scientifique

Ma thèse s'inscrit dans le cadre de la Géométrie de l'Intéraction (Girard) mais vu sous un angle plus moderne de Réalisabilité Linéaire. Dans le cadre de la correspondance preuves-programmes (Chapitre 2), on s'intéresse au lien entre preuves et programmes dans des modèles de calculs variés, typiquement le lambda calcul, mais pas que ! Dans ma thèse, je définis donc des modèles de calculs et j'étudie leurs liens avec la logique.

Mes modèles étendent des modèles existant, notamment les Graphes d'Interaction (IG) de Seiller [6] qui réexpriment la Géométrie de l'Interaction sous forme de système dynamique [7], faisant des liens avec les fonction zeta de graphes.

Les graphes d'interaction sont un modèle de calcul au même titre que le λ-calcul (ils sont techniquement plus expressif, comme un λ-calcul infinitaire)

1 2 3 4

Fig. 5. – Deux graphes d'interaction G et H

ω

1 4

Fig. 6. – Leur exécution $G :: H$

Les programmes sont des graphes par exemple $G = (\{1, 2, 3, 4\}, \{2 \to 1, 2 \to 3, 3 \to 4\})$ en rouge en haut, et $H = (\{2, 3\}, \{3 \to 2\})$ en bleu en bas.

Pour exécuter deux programme, on les recolles et on calcule les chemins maximaux (qui sortent de l'intersection $G \cap H$, donc ici de $\{1, 4\} \to \{1, 4\}$), qui alternent de couleur. On a donc ici dans l'exécution, notée $G :: H$, une infinitée de flêches $1 \to 4$, grâce à la présence d'un cycle alternant $2 \to 3 \to 2$.

1ère problématique

Notez que dans la notion d'exécution présentée plus haut, on exécute deux programmes *entre eux*. Les graphes d'interactions sont en fait *des formes normales* de programmes, des programmes terminés. L'exécution associe à deux formes normales une autre forme normale.

En fait, il est possible de généraliser un peu les graphes d'interactions pour en faire un modèle de calcul dans un sens plus usuel, ou l'on peut directement exécuter un programme. J'ai introduit cela dans ma thèse sous le nom de « Graphes d'Interaction Bicolore » [8].

L'idée est simple: on considère comme primitif non pas les graphes mais les recollement de graphes. Ainsi la Fig. 5 peut-être considérée comme un programme. Ces programmes ont des lieux d'interactions (un peu comme des beta-redex en λ-calcul), ici les lieux $\{2, 3\}$. Une étape d'exécution consiste à retirer un lieu du graphe, et à calculer tout les chemins qui passaient par ce lieu.

Voilà un exemple de lieux $\{2, 3, 4\}$:

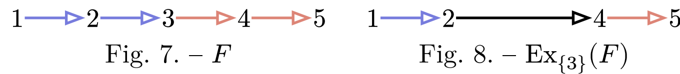


Fig. 7. – F Fig. 8. – $\mathrm{Ex}_{\{3\}}(F)$

On retire le lieu 3 et on calcule les chemins alternant $2 \to 4$. Reste à trouver une couleur naturelle pour cette flêche, qui représente un chemin: on doit retenir qu'il démarrait par une flêche bleue, car sinon on pourrait composer par $1 \to 2$, et qu'il terminait par une rouge car sinon on pourrait composer par $4 \to 5$. Ma solution est simple: autoriser des arêtes bicolore, dont le début et la fin peuvent avoir des couleurs différentes.

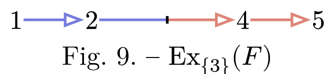


Fig. 9. – $\mathrm{Ex}_{\{3\}}(F)$

Cela permet entre autre aussi d'obtenir une notion d'exécution qui ne soit pas complètement « big step », on obtient la forme normale en plusieurs étapes. Cependant un telle étape peut potentiellement calculer une infinité de chemins (par exemple, retirer 2 dans Fig. 5) on ne peut donc pas qualifier cela de « small step » non plus, j'ai donc décidé de nommer cette exécution « medium step ».

Contexte Scientifique: suite

Ce modèle de calcul des Graphes d'Interaction a été inventé dans le but de « réaliser la Logique Linéaire », c'est à dire de voir certains comportement possible de programmes être décrit par cette dernière.

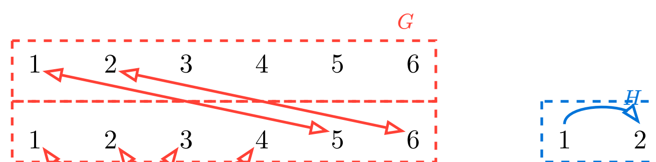
On peut donc compiler/traduire une preuve de Logique Linéaire Multiplcative (MLL) dans ce modèle. On cherche ensuite à obtenir des résultats de correction (voir complétude!) de cette compilation: si une preuve est d'un type A, elle se « comporte comme un A ». Ceci est réalisé formellement via réalisabilité (qui est une formalisation de l'interpétation « BHK »). La réalisabilité définit formellement la notion de comportement.

Par exemple, une fonction $f : A \multimap B$ est telle que pour tout argument $a : A$, $f(a) : B$. On interprète donc la formule $A \multimap B := \{p \mid a \in A, p :: a \in B\}^{\perp\perp}$, ie « l'ensemble des programmes qui étant donnée une entrée de type A, renvoie une sortie de type B ».

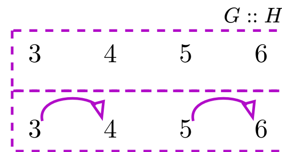
Les Graphes d'Interaction ne peuvent réaliser qu'un certain fragment de LL, la Logique Linéaire Multiplcative Exponentielle (MELL). Pour obtenir LL en entier, il faut regarder des extensions de ce modèle: les Graphes Tranchés et les Graphes Epais.

Conceptuellement, ces deux extensions sont simples:
- Pour les Graphes Tranchés [9], les graphes deviennent des sommes formelles de « tranches »: des boites contenant des graphes. L'exécution se fait en produit cartésien: tranche contre tranche.
- Pour les Graphes Epais [10], on autorise à avoir des arêtes qui vont d'une tranche à une autre. L'exécution se fait toujours tranche contre tranche, mais est un peu plus subtile à définir.

Sur un exemple, si nous exécutons le graphe G contre le graphe H, nous obtiendrons $2 \times 1 = 2$ tranches.



Ce qui donne après exécution le graphe suivant:



2ème problématique

On peut remarquer que dans le graphe $G :: H$, la tranche du haut est entièrement vide. En fait, elle ne sert plus du point de vue de l'intéraction. Ce programme est donc équivalent en un certain sens à celui à une seule tranche (celle du bas).

On aimerais bien pouvoir obtenir directement le graphe à une seule tranche comme résultat de l'exécution et se débarasser de cette tranche superflue.

J'ai proposé dans ma thèse une alternative au Graphes Epais, les « Slider;Graphs »[4]. L'idée est d'obtenir une représentation purement locale des tranches: au lieu d'avoir une tranche qui contient des arêtes, chaque arête vient équipée avec l'information de la tranche d'ou elle viens, et de la tranche ou elle va.

Pour pouvoir etendre mes résultats de manière uniforme avec mes graphes bicolores, il a fallu introduire des arêtes « intermédiaires », comme les arêtes bicolores le sont, qui ont plus d'information que la seule tranche de départ et d'arrivée. J'introduit une notion de « tracker » qui indique ou l'on se trouve dans les tranches des différents graphe. On se retrouve à avoir sur les flèches un ensemble de pré-conditions G qui dit qu'on ne peut prendre l'arête « que si le tracker est bien dans les bonnes tranches » et de post-condition S qui disent « le tracker se déplace pour certaines tranches dans d'autres ». La flêche viens donc équipée de la donnée $S \mid G$. Le nom « Slider;Graph » viens de cette post-condition: « on slide entre les dimensions ». La composition reviens à calculer une forme de « plus petite précondition » et « plus grande post-condition ». Il serait compliqué de donner plus de détails techniques mais voilà une illustration:

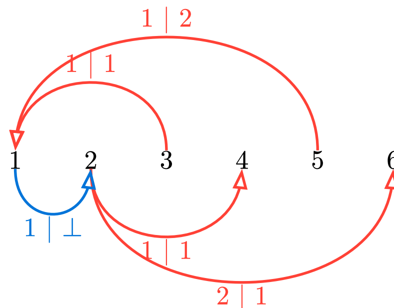


Fig. 10. – G et H vu comme des Slider;Graphs

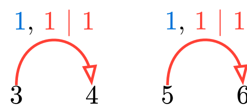


Fig. 11. – $G :: H$

La présence de 2 dans $1 \mid 2$ indique que pour le graphe du dessus, il y a une deuxième tranche. Une fois exécuté, cette tranche disparait, intuitivement car on est passé à travers elle mais du point de vue des « bords » (les extrémités du chemin) on a aucun moyen de le savoir.

Ce modèle résous donc le problème mentionné plus haut. Il n'est cependant pas parfait: le fait de rester local viens avec un problème surnommé « problème des additifs » en GoI: lors de la composition de preuves, il peut y avoir des arrêtes restantes qui ne serviront plus jamais, car invisible du point de vue du typage. On dois donc les quotienter [11], ce qui est encore insatisfaisant. Une solution possible

[4] Le nom fait deux références à la pop-culture: le « ; » viens du dessin animé Steins;Gate et le mot « Slider » vient de la série Slider: Les mondes parrallèles dont mon père me parlait souvent petit. Les deux parlent de voyage à travers les univers parallèles

serait de s'inspirer des travaux d'Hamano [12], qui a une gestion globale des tranches, ce qui donne une forme de « garbage collector » qui efface ces arêtes.

Conclusion de ce premier axe de recherche

J'ai en fait plus généralement suggéré l'idée que les Graphes d'Interaction seraient mieux compris comme des modèles générés à partir de « monoide typés » (calculer des chemins reviens à « composer » des flèches, et les flèches on un type: les lieux). Je propose donc de réinterpreter ces modèles comme étant généré à partir de semi-catégories (la notion formelle de monoide typé). Je peut fabriquer à partir de ces semi-catégories une notion abstraite de programmes dans ce modèle de calcul et prouver les propriétés importantes pour faire de la réalisabilité (associativité de l'exécution notamment). Les constructions expliquées plus haut permette alors d'obtenir des modèles de MALL. Notez que cela peut-être fait à partir de n'importe quelle catégorie ! On peut donc implémenter des programmes en utilisant votre catégorie préférée comme « modèle de calcul ».

Une bonne partie de ce travail est disponible dans ma thèse [8], et en papier est en cours d'écriture sur le sujet, provisoirement nommé « A purely local account of additives » (voir Chapitre 1.3)

Contexte Scientifique: un dernier tour

Il n'est pas entièrement vrai que les Graphes d'Interactions peuvent être utilisés pour réaliser la Logique Linéaire, pour une raison qui paraissait un peu mystérieuse: la disparition d'information géométrique.

Pour assurer les résultats de complétude, on utilise des critères de correction qui assure que le programme viens bien d'une preuve. Notamment, le critère des « Long Voyage » [10] s'exprime comme l'existence d'exactement un cycle dans le recollement de certains graphes.

Malheureusement, dans l'exemple de la Fig. 5, on peut remarquer qu'il n'y a plus de cycle dans $F :: G$. On a notamment pas moyen de savoir s'il était « déjà un graphe infini » ou s'il est comme ça « parce qu'il y a un cycle ». Cette disparition empêche d'obtenir une adjonction entre $\otimes$ et $\multimap$, moralement car $F \otimes G \multimap H$ et $F \multimap (G \multimap H)$ n'ont pas le même nombre de cycles (mais presque, juste que certains ont disparu !). On ne peut donc pas intêrpreter la logique directement !

On a en fait presque l'adjonction via une propriété très proche, la « propriété de 2-cocycle » ou « propriété du trèfle ».

Pour pallier à ce problème, Girard puis Seiller on introduit le concept de « mise », un élément d'un monoide, qu'on adjoint aux programmes (graphes) et qui permet de compter ces cyles. Un programme devient alors un *projet*: une paire (G, a) avec G un graphe et a une mise.

C'est sur les *projets* que se fait vraiment la réalisabilité dans les travaux de Seiller, et la propriété du trèfle assure que la composition des projets soient bein associatives, et donc que cela reste un modèle de calcul.

Problématique

Pourquoi oublier une information pour l'adjoindre à nouveau ensuite ?

Dans ma thèse je propose quelques solutions pour éviter d'oublier les cycles en premier lieu, dont les deux principales sont les suivantes:

- Une assez catégorique, ou l'on évite simplement de supprimer les lieux interne. Je ne la trouve pas entièrement satisfaisante car un peu ad-hoc, cela dit elle rejoint les idées de « calcul medium step » des parties précédentes, qui laisse penser qu'on peut traiter les modèles type GoI et Sémantique des Jeux de manière moins dénotationelle, en considérant le hiding comme une opération de calcul.
- L'autre consiste à considérer comme objet primitif du calcul non plus le Graphe mais sa *réalisation géométrique dirigée* (intuitivement, le dessin du graphe), un espace topologique avec une notion de direction (qui implémente l'asymétrie des flêches). On y remarque que la « vraie » composition est celle du recollement, et dans un monde continu, on a pas besoin de supprimer des vertex: si vous

recollez deux intervalle, le point de recollement deviens une partie de l'intervalle mais se « fond dans la masse ».

On peut quand même retrouver des liens avec les Graphes d'Interaction, en calculant les chemins (dirigés) dans cet espace pour retrouver la forme normale. On peut aussi extraire de l'espace le projet sous jacent en calculant les chemins + les cycles. Toute l'information est là.

Cette dernière solution à été rédigée dans un papier en soumission actuellement [13]. Nous avons espoir de pouvoir faire à terme des liens avec des notions de topologie algébrique et géométrie, comme des ensembles simpliciaux.

Dernières mentions:

J'ai aussi corrigé et simplifié dans ma thèse le modèle dit « des Constellations » ou « Résolution Stellaire » de Eng et Seiller ([14], tiré de la Syntaxe Transcendantale, le dernier projet de Girard) et l'ait comparé a son modèle soeur, les Flots [15]. J'ai défini un calcul de processus (π-calcul) pour essayer de capturer la dynamique de ce modèle de calcul dans un langage plus proche de la communauté.

Ce modèle est intéressant sur plusieurs aspects: il diffère des graphes de par sa non-orientation, et malgré les pistes que j'ai exploré dans ma thèse, ce phénomène est encore mal compris. De plus, il semble correspondre plutôt à des formes d'hypergraphes ce qui généraliserait le modèle actuellement le plus général du domaine, les Graphages [16]. Il est très liés à des modèles de calculs existants, notamment Prolog (Résolution) et les Tuiles de Wang, il y a donc des ponts à établir avec la recherche sur ces sujets.

Nous avons co-écrit tout les trois un article de journal, en révision actuellement, sur le sujet (voir Chapitre 1.3).

3 : Situation actuelle

Après avoir effectué une thèse sur des sujets assez théorique, j'avais envie de travailler quelques temps sur des applications plus concrètes, et d'étendre un peu mes perspectives de recherche. Je souhaitais aussi en apprendre plus sur la Sémantique des Jeux, qui est un domaine « soeur » de la GoI/ Réalisabilité Linéaire ([17])

J'ai eu la chance d'obtenir un post-doc parfait dans cette optique: j'aide à redéfinir les fondations théoriques sur lequel repose l'assistant de preuve pour la cryptographie Squirrel [18], en utilisant de la sémantique des jeux !

3.1 - Mon projet dans Squirrel

Cet assistant est spécialisé dans la preuve de sécurité de protocoles cryptographiques. Historiquement, il se base sur la logique CSSA [19]. Les agents (programmes) de ces protocoles par des symbole de fonction du premier ordre, par exemple $\mathrm{hash}(-)$, un protocole était alors encodé comme un terme, par exemple: $(\mathrm{if}\ \mathrm{atk}_0() == \mathrm{sk}\ \mathrm{then}\ 1\ \mathrm{else}\ 0)$.

Toute cette syntaxe étant finalement interprété en terme de machine de turing polynomiale probabilisite, pour pouvoir donner du sens à des expressions telles que: $(\mathrm{if}\ \mathrm{atk}_0() == \mathrm{sk}\ \mathrm{then}\ 1\ \mathrm{else}\ 0) \sim 1$ qui veut intuitivement dire qu'un « attaquant » raisonnable ne peut pas trouver une clé à la laquelle il n'a pas accès, ce qui se prouve ensuite dans l'assistant via des règles de Déduction Naturelle (et sous des hypothèse sur la fonction de hachage)

Le logiciel et sa théorie ont beaucoup évolué depuis, et dans le but d'ajouter de l'ordre supérieur et obtenir des programmes plus proche d'un vrai language de programmation, ces termes ont été remplacés par des λ-termes. Une fonction de hash avec clé secrete sk devient ainsi $\lambda m.\ \mathrm{hash}(m, \mathrm{sk})$.

De plus, les preuves Squirrel ne garantissent en fait pas une indistinguabilité dans le « Modèle Computationel », ce que les Cryptographes attendent habituellement, mais une forme de garantie plus faible: [18], ou l'attaquant ne choisi pas la durée de l'interaction (qui peut être cependant aribtrairement longue). Cela garantie par contre des preuves plus simple, notamment utilisant le principe *d'induction sur la trace d'exécution* qui n'est pas valide dans le modèle computationel.

La sémantique de l'assistant n'a pas évolué aussi vite est reste « bloquée » sur une interprétation très ensembliste et peu appropriée pour du λ-calcul. Ainsi, mon travail actuel consiste à redéfinir un lambda calcul pour Squirrel, et lui donner une Sémantique des Jeux. Ce type de sémantique modélise les λ-termes comme des stratégie dans un jeu qui oppose « Player » (le programme) et « Opponent » (le contexte de ce programme).

Définir une notion de stratégie probabiliste polynomiale est en fait assez simple, et était déjà présente antérieurement [20], mais utilisée dans un contexte différent (le but était d'obtenir des résultats négatifs sur la possibilitée de cryptographie d'ordre supérieure, ou les données envoyées pourraient etre non plus seulement des messages « binaires » mais des programmes vu comme des boites noires. Notez

que ces résultats négatifs n'empeche pas l'utilisation de programme d'ordre supérieur en cryptographie, on peut tout à fait faire écrire un protcole en OCaml).

Nous avons donc repris et étendu leurs travaux, en définissant un λ-calcul approprié comme langage de programmation pour définir des protocoles cryptographiques. Ces programmes sont interprétés comme des stratégies, et nous utilisons une interprétation alternative du $\sim$ mentionné plus haut comme une indistinguabilitée par des « opposants » probabiliste polynomiaux.

L'intéret de la sémantique des jeux simple (« à la Hyland / AJM », [21]) est notamment que les formules qui portent sur la trace d'exécution s'y interpretent tout naturellement, ce qui permet de garder un langage très proche du Squirrel d'origine.

Une fois cela accompli, un nouveau problème apparait: beaucoup de preuves Squirrel reposent sur l'induction sur la trace, qui n'est vraie que si $\sim$ est un $\sim$ « faible/non-uniforme » dans l'analogie présentée plus haut. On essaie donc d'importer de nouvelles règles de déduction provenant de techniques classiques du λ-calcul, tel que les bisimulation et les relations logiques pour obtenir des preuves d'équivalences « fortes ».

4 : Projet de Recherche

J'ai plusieurs pistes de recherches que j'aimerais explorer à l'avenir, que j'essaie de présenter succintement ici

4.1 - Court terme

Ces projets me semblent réalisables sur quelques années (< 3 ans), il faudra bien sur faire des choix sur lesquels je priorise.

Je tiens à attirer l'attention sur le fait que plusieurs d'entre eux peuvent s'interfacer avec différentes domaines: système dynamiques, preuves circulaires, calcul parallèle, cryptographie, etc… J'aimerais avoir la chance de pouvoir interagir avec des chercheurs de ces différents domaines

Graphes d'Interactions: Extension Naturelles

J'ai amélioré dans ma thèse (Chapitre 2.3) le modèle des additifs de Seiller [11], mais il y a encores de nombreuses extensions possibles qui peuvent être investiguées à court termes, entre autres:

- Définir des Graphes d'Interaction pour μMALL, la logique linéaire avec plus petit et plus grand point fixe. Les preuves circulaires sont plus expressives que les preuves habituelles et établissent des liens avec la théorie des automates, ce qui en fait un sujet d'intérêt remarquable (voir EPIT 2025 pour voir les liens d'intêret). Il semble justement qu'un enjeu pour mieux comprendre ces logiques soient des les regarder sous l'angle de la Géométrie de l'Interaction (conclusion de la thèse [22]). Ce serait un pas dans cette direction.
- Définir des Hypergraphages: un équivalent « graphage » du modèle des constellations de la Syntaxe Transcendantale. Le but étant de tester et potentiellement étendre la définition de modèle de calcul abstraite avancé par Seiller dans [23]. Il essaie en effet de trouver une notion abstraite de modèle de calcul qui permettrait d'englober en une seul objet des définitions qui ont a priori l'air différentes (Machine de Turing, λ-calcul, π-calcul etc…). Sa définition est inspirée des IG, mais il se pourrait que les Hypergraphages soient plus généraux et donc pas capturé, ce qui apporterait alors une notion plus générale de modèle de calcul!
- Etablir des liens plus clairs avec la topologie algébrique dirigée: les calculs effectués par les IG, et leurs généralisation a LL, les graphages [16] sont en fait des calculs de chemins dans des espaces topologiques orientés, espaces obtenus par recollement, comme mentionné dans mon papier en cours de publication: From Geometry to Interaction[13]. Peut-on généraliser ces résultats à d'autres espaces que des réalisations de graphes ?
- Les modèles de graphes d'interaction sont traditionellement des modèles de la Logique Linéaire. On pourrait essayer de les étendre pour réaliser des extensions directe de LL. Par exemple LL est décrite dans un cadre commutatif. Il y a pourtant la possibilité de définir des comportements non

commutatifs: $A < B := \{p \mid \neg(e \in p, s(e) \in B, t(e) \in A)\}^{\perp\perp}$, on peut donc se poser la question de la possiblité de réaliser aussi la Pomset logic [24], ou d'autres logiques non commutatives. On pourrait aussi étudier la réalisabilité de DiLL: quel genre de programme implémente la différentiation d'un autre programme?

Graphes d'Interaction et types dépendant

J'ai observé dans ma thèse que les types d'un modèle de calcul au sens de Seiller forment eux même un modèle de calcul. En terme de théorie des type, cela devrait pouvoir (entre autre) donner une interprétation calculatoire à la substitution, usuellement méta-théorique $\Pi(x : A)B[x] :: (a : A) \to B[a]$ mais qui, sur un ordinateur, est bien implementée par un programme ! Il serait intéressant de dévelloper le premier (à ma connaissance) modèle de GoI pour la théorie des types, si possible dépendante, et de voir si cette observation permet d'obtenir une interprétation plus précise des règles usuelles.

Notamment, la question de la théorie de trouver une théorie satisfaisante des types dépendant linéaire n'est toujours pas résolue: un accompte propre avait été effectué dans [25], mais la solution est incomplète: le cas de la dépendance d'un type non linéaire en un terme linéaire reste élusive, et de nombreux travaux sont effectués dans une recherche de solution [26], [27], [28], [29]. Ce travail serait un petit pas dans cette direction.

Sémantique des Jeux

Les modèles de jeux historique enforcaient une notion d'alternance entre joueur et opposants. Cela n'est plus le cas désormais dans les jeux qui étudient le parallélisme: les jeux conccurent [30], qui sont une forme de jeux modernes et « à la mode » dans le domaine.

Les modèles de GoI type IG sont eux toujours contraints par l'alternance. On pourrait alors essayer de dévelloper un modèle de Graphes d'Interactions conccurent, sans cette notion d'alternance. Une possible inspiration pourrait venir de vieux travaux du temps de la ludique: [31]. Il serait intéressant de voir si ces réseaux de preuves sont encore « à jour » aujourd'hui, typiquement en essayant d'y interpréter IPA (Idealised Parallel Algol).

Plusieurs questions techniques se poseraient pour une telle extension, notamment ce qu'il adviendrait du critère de correction (long chemin), qui est à priori fortement dépendant de cette alternance. Cela mènerait à un style de GoI nouveau.

Une autre question qui se poserait naturellement est que les modèles de constellation n'ont pas vraiment de notion d'alternance équivalente, serait il possible d'interpréter directement IPA ?

Nouvelles Fondations pour l'assistant de preuve Squirrel

A la fin de mon post-doc (voir Chapitre 3), nous devrions être dans la situation suivante:

Une nouvelle sémantique, basée sur les Jeux, viens d'être intégrée. Mais ce n'est qu'un tout début d'un renouveau pour Squirrel: le nouveau λ-calcul qui modélise les protocole est minimal, les règles de relation logique aussi. On a crée un nouveau « coeur » mais il faut maintenant en faire un vrai langage

- On devrais pouvoir étendre ce coeur minimal: ajouter de nouveaux constructeurs (ex: additifs), de nouveaux types de données, des nouvelles propriétés du langage (état locaux, calcul localement parallèle) etc… Il y a donc un gros travail syntaxique à faire: étendre le calcul et sa sémantique petit pas etc… Ainsi qu'un travail sémantique qui suit: définir les arènes de jeux etc…

  Tout ce travail a été fait dans des jeux de types Hyland/AJM, mais beaucoup de travaux modernes sur les jeux sont dans un style plus HO. Il faudrait donc potentiellement adapter les résultats récentes dans le cadre Hyland, ce qui nécessite un certain travail théorique.
- Il faut procéder à un interfaçage avec les anciennes preuves de Squirrel, car le nouveau langage, bien que le plus proche possible de l'ancien, n'est pas exactement le même.
- Il faudrait dévelloper de nouvelles heuristiques et méthodes de recherche de preuves pour remplacer les anciennes, vu l'introduction de nouvelles règles de déduction basées sur des relations logique / les bisimilarités applicatives, pour pouvoir prouver des équivalences sans utiliser l'induction sur la trace.

Evidence Frame pour la Réalisabilité Linéaire

Les techniques de réalisabilité dans différents modèles sont assez similaires, il y a donc eu des tentatives d'unifier les différentes notions dans la littérature. Une des notions fondamentale en Logique Catégorique est celle de Topos, une catégorie ou l'on peut interpréter les formules de la théorie des ensembles notamment, et il se trouve que des modèles de réalisabilité on peut construire un Topos. En fait, cette construction peut-être factorisée: on peut construire un objet « plus simple », un Tripos, et tout les Topos de réalizabilité sont en fait généré par des Tripos. Cette construction peut elle encore être factorisée a travers un objet beaucoup plus simple: Miquel a ainsi introduit le concept d'algebre implicative [32], qui capture de manière abstraite et simple une notion de modèle de réalisabilité pour la logique classique et intuitionniste. Cette notion a été récemment étendue par Lucquin, Seiller et Pelissier [33] en algèbre implicative lineaire pour capturer aussi les modèles de réalisabilité linéaire. Parallèlement, une autre généralisation des algèbre implicative et plus générale à été conçue par Cohen et al, les Evidenced Frame [34] pour pouvoir permettre notamment l'ajout d'effet dans les modèles de calculs considérés. Il serait donc intéréssant d'adapter les travaux récent sur les algebres implicative linéaire dans le cadre des évidence frame pour obtenir une théorie élégante est très générale de la réalisabilité.

4.2 - Moyen Terme

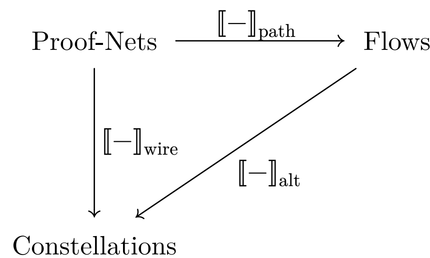
La résolution de ces projets est moins certaine, mais semble possible. Reste toujours l'option de faire d'autre projets court termes si ceux de cette section se révèle trop hardu.

Graphes d'Interactions, encore

Il y a d'autres travaux dans le cadre des IG que je pourrait poursuivre, notamment:

- S'inspirer des travaux de [12] pour résoudre le problème des additifs dans le cadre des IG. On pourra notamment éclaircir le concept de « location » qui est présent dans les travaux de recherche sous trois aspect différents: les locations en GoI, en Logique Linéaire Indexée et en Sémantique des Jeux.

- Dans ma thèse [8] je montre comment on peut définir des modèles similaire aux Graphes d'Interaction sous forme de (semi-)catégories et foncteur libres, ce qui permet d'obtenir directement énormément de modèles (notamment, chaque catégorie induit un modèle de calcul ainsi !). Ce travail reste limité car on se trouve avec un niveau d'expressivité similaire à celui des graphes d'interactions, on devrait essayer de l'étendre pour les cas exponentiels, et notamment essayer de définir de manière plus abstraite l'exponentielle de Seiller, ce qui rendrait ses travaux plus accessibles à la communauté.
- On peut aussi chercher à résoudre le conflit entre Flot et Constellations:



Il existe deux compilation différentes des preuves MLL dans le modèles des Constellations:

- Une directe, dans un style « Danos-Regnier »
- Une passant par les flots, dans un style « Long-Chemin »

Ces deux interprétations semble différentes mais liées, et il n'est toujours pas clair de pourquoi plusieurs inteprétations de la même preuves sont possibles et des différences profonde entre les deux. Résoudre cette tension permettrait de mieux comprendre les différences de nature entre ces deux modèles de calcul.

Contenu Calculatoire du Premier ordre

La logique du premier ordre à un rôle très particulier en mathématique:

- C'est la « meilleure » logique d'un point de vue métathéorique: la seule qui soit compacte et vérifie Löwenheim-Skolem descendant (théorème de Lindström). La déduction naturelle / le calcul des séquents sont complet. La théorie des ensembles est d'ailleurs une théorie du premier ordre ! (Même si en un sens elle fait s'effondrer la hiérarchie)
- Elle n'a a priori aucun contenu calculatoire ([35]): d'un point de vue correspondance preuve-programme, il existe un foncteur d'oubli du calcul des prédicats du second ordre vers Systeme F, qui préserve le contenu calculatoire et qu'on peut donc « inverser », ce qui permet notamment de déduire la terminaison du dit calcul des prédicats.

Ce résultat est étonnant et un peu décevant, un des but premier de la réalisabilité étant de trouver du sens calculatoire aux axiomes des mathématiques. De plus la théorie des types donne bien une forme de contenu calculatoire a des propositions du premier ordre telles que $\Pi(x : \mathbb{N}).(x + 2 = 2 + x)$ ce qui semble entrer en contradiction avec la remarque évoquée plus haut.

Girard à proposé une ébauche dans ses travaux récents [36] de sémantique qui donnerait un contenu calculatoire à la logique du premier ordre qui mériterait d'être exploré. Il serait aussi intéressant de résoudre la tension entre théorie des types et logique du premier ordre vu sous l'angle Curry-Howard.

Une Logique Lineaire Différentielle non symétrique

Mes travaux sur Chapitre 2.1, étendu ensuite par Koleilat [37] pour créer une Logique Linéaire Différentielle avec Types Dépendant, offre une belle présentation des modèles de DiLL.

Les catégories modèles de DiLL sont des catégories monoidales *symétriques*, autrement dit la règle d'échange y est admise.

Il existe d'autre part des variantes de la logique linéaire non symétrique, notamment la Logique Pomset de Rétoré [38]. Cette variante est notamment utilisée pour étudier les grammaires catégorielles de Lambek [24], des outils mathématiques introduit par Lambek avec des applications à l'analyse du langage [39].

Il serait intéressant de dévelloper une variante non commutative de la logique linéaire différentielle, tant du point de vue syntaxique/théorie de la preuve, que sémantique en relachant les hypothèses dans la définition catégorique.

On pourrait alors envisager d'utiliser ces catégories pour l'étude du langage. Il serait intéressant de voir ce à quoi correspondrait la différentiation dans ce cadre. Dans le cadre du λ-calcul, on peut effectuer une expansion de taylor: convertir un programme en une somme de sous-approximation tel que le terme en x^n utilise exactement n fois son argument [40]. Peut-être une conversion similaire est-elle possible pour les phrases ?

Interprétation abstraite et Réalisabilité?

L'interprétation abstraite est une technique d'analyse de programme, ou l'on essaye d'extirper de la syntaxe des informations sémantique. On ne peut malheureusement pas extirper la sémantique dénotationelle exacte d'un programme, puisque cela permettrait de résoudre le problème de l'arrêt. On doit donc procéder a des (sur-)approximations du *comportement* du programme.

La réalisabilité se pose aussi la question de comportement des programmes. Le théorème principal (l'adéquation) dit qu'un programme qui est bien typé (une contrainte syntaxique) se comporte comme on l'attend (une sémantique). Notamment, on peut voir les système de typage comme de l'interprétation abstraite [41]. Traditionellement, la réalisabilité ne se préoccupe que de typage « simple », par exemple $A \multimap B \coloneqq \{p \mid p :: a \in A, b \in B\}^{\perp\perp}$. On peut alors regarder le comportement $\mathbb{B} \multimap \mathbb{B}$ avec $\mathbb{B}$ les booléens. Mais même si à ma connaissance cela n'a jamais été fait dans ce cadre, rien n'empeĉhe de définir des types de données bien plus précis, par exemple $2\mathbb{N} \coloneqq \{2n \mid n \in N\}$ et de se poser des questions telle que $f \in \mathbb{N} \multimap 2\mathbb{N}$?

Y répondre reviens à résoudre des problèmes similaires à ceux de l'interprétation abstraite: on a $f : n \to 2n$ et $g : n \to 4n$ qui sont toutes les du type vu plus haut. Arriver a obtenir cette information nous donne une sur-approximation du comportement de f et g qui peut nous donner des garanties statique forte sur notre programme. Evidemment, ce problème n'est a priori pas résoluble de manière exacte.

Pourtant se pose réalisabilité linéaire se pose la question de comment capturer *exactement*, et de manière *calculable*, un type. Cela est fait à travers une notion de « tests »: un ensemble fini de programmes qui en intéragissant avec un autre programme, garantissent son typage (et ce de manière finitaire !). Cela est possible pour des types « simple », tel que ceux de MLL. Grace à cette représentation finitaire des types, on peut alors décider beaucoup de choses: l'appartenance a un type, les intersections de type etc… Ce qui rapelle un peu les contraintes que l'on attend d'un domaine abstrait (une version « calculable » des opérations usuelles). On peut donc dans certains cas obtenir des résultats *exacts* d'interprétation abstraite. Mais il faut des contraintes fortes sur la forme du programme. Cependant, toute information conservée est bonne à prendre, et on pourrait, notamment dans un langage proche de la logique linéaire tel que Rust, espérer être capable en combiant plusieurs domaines d'interprétation abstraite, des exacts tiré de LL et des non-exact, extirper le plus d'information possible.

Le passage à des types plus compliqués, tels que le second ordre, n'est pas possible pour exactement les même raisons que celle vue en interprétation abstraite. Girard propose une « solution » a ce problème [42], qui semble intuitivement ressembler a une approximation (une sous-approximation ici), qui est une garantie de la forme « si on passe les tests, on marche bien », mais on ne peut pas garantir que tout le monde passera les test. Il serait intéressant de voir si des liens existent entre les deux approches, qui semblent duales. Cela nécessiterait cependant l'expertise d'un chercheur en interprétation abstraite.

4.3 - Long Terme

Je dévellope ici une ébauche de recherche sur du très long terme, il n'est pas clair du tout qu'elle puisse aboutir, mais les progrès fait en la suivant seront intéressant pour la correspondance preuve-programme et la réalisabilité de manière générale

Une Réflexion sur les Assistants de Preuves

Les assistants de preuves reposent majoritairement sur des langages de programmation (en général, le λ-calcul), notamment HOL & Isabelle (une théorie des types assez faible), Rocq, Lean et Agda (CiC et MLTT), Dedukti / Lambda-PI (sur la théorie Lambda-Pi), à l'exception notable de Mizar qui repose sur la théorie des ensembles.

Ces langages de programmation ne sont souvent pas inter-opérables, et c'est la raison de la naissance de Dedukti (Chapitre 1.3), qui sert de « hub » de preuves, et tout ceci nécéssite des compilations depuis et vers Dedukti avec ces assistants de preuves.

On peut se poser la question de pourquoi faire reposer nos prouveurs sur le λ-calcul et pas sur un autre modèle de calcul ?

La raison profonde de l'utilisation de ces programmes est la garantie de correction qu'ils apportent: la normalisation d'un λ-calcul assure la correction de la logique de son système de type: ainsi, les preuves sont obtenues en combinant des programmes, et la connaissance du fait que ces programmes normalise assure qu'ils ne sont pas des preuves du faux: typiquement, en calcul des séquents, la seule manière d'introduire le faux est la règle de coupure (moralement, l'application d'une fonction a un argument), et la normalisation consiste à éliminer ces coupures.

Il y a donc traditionellement deux couches dans un système de déduction logique:

- Une couche des propositions (typage)
- Une couche des programmes (justifieur): ils permettent de justifier la validité des règle de déduction, par exemple l'existence d'un programme **tens** : $(\Gamma \multimap A) \otimes (\Gamma \multimap B) \multimap (\Gamma \multimap A \otimes B)$ dans un modèle de calcul donné (dont on sait la terminaison) justifie la validité de la règle d'introduction du tenseur de la Logique Linéaire: c'est le programme qui transforme deux programmes de types $(\Gamma \multimap A)$ et $(\Gamma \multimap B)$ en un nouveau programme de type $(\Gamma \multimap A \otimes B)$

Lorsqu'on écrit une preuve en déduction naturelle, dans Rocq ou Agda etc… On combine en fait sous le capot nos axiomes en les branchants à travers ces programmes justifieurs, qui servent de « liens ».

Une alternative (disons, idée complémentaire) à la solution Dedukti que je mentionne dans ma thèse serait de ne pas combiner directement les programmes, mais d'ajouter un niveau d'abstraction intermédiaire entre ces deux couches, ce qui revient à utiliser comme « hub » un système de déduction « à la Hilbert », la logique des combinateurs.

Un tel prouveur aurait donc trois couches:

- Une couche de syntaxe **S** des propositions, ou l'on déclare les opérateurs que l'on souhaite utiliser (par exemple $\otimes$)
- Une couche de preuves **P** dans un système Hilbert minimal (la seule règle est le modus ponens), ou l'on déclare les combinateurs/règles logiques que l'on souhaite utiliser, par exemple $\otimes_{\text{intro}} : (\Gamma \multimap A) \otimes (\Gamma \multimap B) \multimap (\Gamma \multimap A \otimes B)$. C'est ici que sont faites les preuves en combinant les différents combinateurs.

  Dans un tel setting, une logique est simplement un ensemble de combinateurs. Une preuve peut être faite dans une logique donnée: on déclare simplement quels combinateurs on s'autorise au début du fichier. Tout le reste n'est que de la combinatoire de ces combinateurs.
- Une couche de justificateur **J**: la donnée d'un langage de programmation, une preuve de sa terminaison, et de programmes qui réalisent (qu'on peut compiler vers) ces combinateurs.

En pratique, une quatrième couche serait extremement commode: une interface pour écrire des preuves plus facilement, car les systèmes de Hilbert ne sont pas très pratique d'utilisation pour des utilisateurs humains. En pratique cette couche peut-être la même que la dernière: typiquement, Rocq peut servir d'interface et de justificateur.

Une autre chose à noter et qu'on peut justifier des règles dans un langage qui ne termine pas, mais il faut pouvoir garantir que le fragment du langage correspondant a nos justificateurs termine. C'est ce qui arrive en réalisabilité linéaire, et cette garantie est obtenue via le critère de correction de la logique linéaire. Un avantage de cette approche est qu'on aurait potentiellement jamais besoin de changer de système de justifieur: vu que ce modèle de calcul ne normalise à priori pas, on sait qu'il est inconsistent, mais il doit contenir une infinité de fragment consistant de plus en plus grand (qu'il faudrait caractériser).

Il me semble important de justifier de l'intêret d'un tel système, avec un « cas pratique » fictif:

> Imaginons que j'ai effectué une preuve du théorème de pythagore dans **P**. Je souhaite m'en servir pour faire une preuve de la terminaison de Rocq. Malheureusement, mon langage de justifieur est actuellement CoC (la base de Rocq), je sais donc que je ne peut pas prouver la terminaison.
>
> Je décide alors de changer de justifieur, et de prendre Rocq + une hiérarchie d'Univers (ce qui est déjà plus proche du vrai Rocq). Je dois alors à priori re-justifier toutes mes règles (en pratique, une automatisation peut-être possible via une compilation, ici l'identité qui envoie CoC dans CoC + Univers, et je n'ai qu'a justifier les nouvelles règles).
>
> Ma preuve a été effectuée dans **P**, elle est donc indépendante des justifieurs: c'est toujours la même preuve, et je peux la réutiliser.

On peut alors réfléchir aux avantages d'une telle approche:

- Le paragraphe précédent montre que les avantages de Dedukti sont toujours présent: on a bien un hub central dans lequel toutes les preuves s'accumulent. Fondamentalement, justifier une logique reviens à faire une compilation d'un langage vers **P**, ce qui est le premier pas à faire pour importer des preuves en Dedukti. On se retrouve aussi avec un seul langage interface pour les preuves.
- La compilation en question est généralement très simple: traduire la déduction naturelle ou le calcul des séquents en combinateur est direct. Les systèmes de Hilbert ne sont pas contraints: Dedukti est basé sur une théorie des types, ce qui fait qu'un système comme Mizar qui est basé sur une théorie des ensembles est difficilement compilable dans Dedukti. Ici, la seule règle est le modus ponens (qui est présent dans la grande majorité des systèmes logiques, pour ne pas dire tous), et donc on peut y encoder facilement des logiques.
- Notamment, on doit pouvoir encoder facilement des logiques alternatives, qui n'ont pas forcément de calculs des séquents ou déduction naturelle (logique modales, temporelles etc…)

- La recherche de preuve se fait directement combinatoirement en essayant de combiner des combinateurs, on devrait pouvoir plus facilement envisager de faire de la recherche automatiques de preuves traditionelle, ou d'entrainer des IAs pour cela dans un tel système. Et pour obtenir des données d'entrainement, il suffit de compiler toutes les preuves de tout les prouveurs existants dans ce langage.

⇒ Il y a de nombreuses difficultés à explorer réaliser un tel projet:

- Comment bien gérer l'ordre supérieur (dans le sens HOL): l'ordre supérieur est en effet un peu « étrange » d'un point de vue logique: la puissance effective d'une formule du second ordre $\forall A.P[A]$ dépend de la logique ambiante !

  Prenons un exemple: $\forall A.A \multimap A$.

  - Si je me place dans MLL + 2nd ordre, je peux remplacer A par $\mathbb{N} \otimes \mathbb{N}$, ce qui me donne une preuve de $\mathbb{N} \otimes \mathbb{N} \multimap \mathbb{N} \otimes \mathbb{N}$ mais pas par $\mathbb{N} \& \mathbb{N}$ qui n'est pas une formule de MLL.
  - Cette même preuve dans MALL + 2nd ordre me donne une preuve de $\mathbb{N} \& \mathbb{N} \multimap \mathbb{N} \& \mathbb{N}$

  Cette dépendence au système logique, qui est formellement exprimée par le fait que les connecteurs du second ordre n'ont pas la **propriété de la sous formule** donne un statut très particulier au second ordre (c'est d'ailleurs la raison profonde de l'incomplétude), qui rend difficile les méthodes traditionelle de justification par réalisabilité de manière « externe à un système logique ».

- La gestion des types dépendants: les types dépendant sont quelques peu mystérieux, car ils « mélangent les couches »: les programmes, qui servent habituellement de justifieurs, apparaissent dans la logique elle même ! Ce qui rend difficile l'inter-opérabilité puisque les couches sont mélangées ! Il n'est pas clair de comment ils pourraient être gérés, mais la problématique est en fait assez liée à la précédente.

Autrement dit, cette direction de recherche en est encore à ses balbutiements, mais elle lie entre elle de nombreux domaines de la logique ce qui la rend propice à des nombreuses collaborations.

5 : Enseignements

Vu mon parcours scolaire et de recherche, on pourrait penser à une préférence naturelle pour la théorie, et c'est vrai pour ma recherche, mais en enseignement je préfère enseigner des choses plus concrètes d'ou mon choix par exemple d'enseigner l'algèbre linéaire deux ans, car elle servira aux étudiants peu importe leur parcours ensuite. Je n'ai cependant pas de réticence à enseigner des choses théoriques non plus.

	Heures	**Niveau**	**Domaine**	**Année**
Informatique pour les Mathématiciens	14h **TD**	M1	Informatique/Math	2022-2023
Logique (ENS-Rennes)	19h30 **TD**	M1+L3	Informatique/Math	2025-2026
Kholle (L2 MPCI)	10h	L2	Math	2022-2024
Architecture des Systèmes	9h **TD** + 12h **TP**	L2	Informatique	2024-2025
Programmation Java avancée	44h **TP**	L2	Informatique	2024-2025
Algèbre Linéaire	27h **TD** + 9h **CM**	L1	Math TC[5]	2022-2024
Outils Mathématiques	30h **CM**	L1 en 2 ans	Math TC[5] + Pédagogie	2023-2024
3 Stages Hippocampe	36h **TD** (54h réelle)	4ème-1ère	Pédagogie Math/Info	2022-2024
⇒ **Total**: 210h30	79h30 **TD**	56h **TP**	39h **CM**	36h **Péda**

Je souhaite attirer l'attention sur mon profil Math/Info qui peut-être profitable: je sais m'adapter aux deux publics et j'aime faire des ponts dans mes enseignements. Je voudrais aussi insister sur l'intêret que je porte à la pédagogie: j'ai choisi de faire des enseignements à des élèves de niveau plus faible (notamment les deux dernières ligne du tableau) parce que la pédagogie me tiens à coeur.

5.1 - Détails des Enseignements Effectués

J'ai fait ma thèse dans un laboratoire de Mathématique, et donc ai du enseigner en étant rattaché a l'UFR de Mathématiques pendant 3 ans. Étant Informaticien de formation, j'ai souhaité acquérir de

[5]Tronc Commun

l'expérience en enseignement de l'Informatique, et j'ai obtenu une autorisation la 3ème année pour enseigner des cours de l'UFR d'Informatique (en restant attaché a l'UFR Maths).

J'ai donc enseigné en thèse, à raison de **64h par an**:
- **2 ans les Maths**
- **1 an l'Informatique**

J'ai cependant **beaucoup enseigné en tronc commun math/ informatique**, même dans les années « Maths ».

Finalement, j'ai décidé d'**enseigner en post-doc en supplément de mon contrat.**

⇒ Année 1 (Math + tronc commun)

- **2 stage hippocampe (Vulgarisation), 2x(3x6) heures (compte pour 12h TD):** On donne des problèmes « ouvert » à résoudre a des collégiens / lycéens et l'on essaye de leur faire découvrir le processus de recherche. Par exemple, on leur donne des carrés assemblés sous forme de « pyramide » et on leur demande de chercher une formule pour trouver la somme triangulaire. On ne leur donne pas la solution directement, il faut les aiguiller quand ils bloquent
- **Informatique pour les Mathématiciens (M1) (14h TD):**

  Le principe est d'enseigner les bases du Python et de l'algorithmie aux mathématiciens. On leur découvrir des algorithmes, prouver leurs corrections, faire des liens avec les maths que eux connaisent, je prendrais pour exemple le fait que l'algorithme de division euclidienne et « le même » que celui d'orthonormalisation de Gram-Schmidt en utilisant les bonnes abstraction.
- **Algèbre Linéaire (L1) (18h TD):**

  Les bases de l'algèbre linéaire, l'algorithme du Pivot de Gauss, les Matrices, changement de bases, les définitions d'un Espace Vectoriel etc…
- **Kholle (a la FAC), sur les transformations et isométries en 2D (L2) (6h)**

⇒ Année 2 (Math + tronc commun)

- **1 stage hippocampe, 3x6 heures (compte seulement pour 12h):**

  Voir l'année précédente pour la description.
- **Algèbre Linéaire (L1) (9h CM / 9h TD):**

  Après avoir été seulement chargé des TDs en première année, j'ai commencé à donner une partie du cours, notamment les chapitres sur comment calculer à l'aide d'une matrice le noyau et l'image d'une application linéaire (ainsi que des bases de ces espaces)
- **Outils Mathématiques (L1 en 2 ans, 1ère année) (30h CM):**

  Ma première classe en tant que prof complet. J'ai enseigné les bases des nombres complexes et des équations différentielles linéaire. Ce n'était pas facile: cette L1 est une L1 spéciale pour les élèves en difficulté a la sortie du lycée, et qui doivent rattraper leur retard (on leur fait donc faire la L1 en 2 ans pour plus prendre notre temps). Heureusement, le nombre d'élève est limité donc on peut avoir une approche plus personalisée de l'enseignement.
- **Kholle (a la FAC), sur le calcul différentiel (L2) (4h TD).**

⇒ Année 3 (Info)

- **Programmation Java avancée (L2) (44h de TP):**

  On reprend rapidement les bases de Java et de la programmation impérative, puis explique toute le potentiel du langage en détail. On y découvre notamment la Programmation Orientee Objet

accompagnés de quelques Design Pattern, mais aussi comment écrire du code dans un style plus Fonctionnel (iter, map etc…). Il y a un petit peu de lambda calcul en bonus a la fin.

L'accent est vraiment mis sur comment écrire du code très propre, clair et concis, sur comment refactoriser efficacement. Les TP consistent en d'abord des exercices pour revoir les notions du cours, puis des projets logiciels à réaliser (1 par TP) qui consistent en étendre une base de code donnée pour établir des fonctionalitées demandées. On pousse ainsi les élèves a se rapprocher du vrai travail de dévellopeur.

- **Architecture des Systèmes (L2) (9h TD, 12h TP):**

  Ce cours explique a partir de zéro comment fabriquer (théoriquement, pas physiquement) un ordinateur: on part des portes logiques, on ajoute des transistor pour la mémoire, on explique les machines de Mealy/Moore (des transducteurs, mais sans trop rentrer dans la théorie des automates), on implémente la fonction de transition de ces machines via des portes logiques dans un logiciel. Tout a la fin, on réalise un véritable ordinateur: on réalise une machine qui agit comme un vrai cpu, charge une fonction de transition (qui n'est plus codée en dur mais deviens une donnée) et execute le programme.

  En TD, on apprend a faire les « calculs » (compiler/traduire un programme en assembleur « simple » en un automate, expliciter la fonction de transition d'un automate etc…). On utilise ensuite ces compétences en TP sur ordinateur pour créer ces programmes sur logiciel avec les portes logiques.

⇒ Année 4, en Post-Doc (Math-Info)

- **Logique (L3/M1 Math-Info ENS-Rennes):**

  Une introduction à la logique pour les L3 Info et les M1 Maths-Info. On y voit entre autres les liens entre calcul des séquents et déduction naturelle, la correction et complétude de la logique du premier ordre, l'élimination des quantificateurs, la méthode de résolution pour générer des preuves, la théorie des modèles fini (Jeu d'Ehrenfeucht-Fraïssé). J'ai pu reprendre certains TDs d'années précédentes, mais j'ai du aussi en créer par moi même. En général c'était un entre deux, adapter d'ancien TD + ajouter des exercices.

5.2 - Orientations Pédagogiques

Maths-Info

J'aime beaucoup faire des ponts entre les maths et l'informatique pour mes étudiants; et ce dans les deux sens, par exemple:

- Des maths vers l'informatique, j'aime donner une saveur plus entre-deux à l'Algèbre Linéaire, je recommande toujours Interactive Linear Algebra à mes étudiants pour les aider à la visualisation, et j'aime leur montrer des applications réelle (cela permet souvent un regain d'intêret pour la matière), par exemple l'utilisation d'hyperplan comme classifieur en machine learning.
- De l'informatique vers les maths, j'aime proposer des TPS de programmation avec des exercices plus mathématique: dessiner des figures géométriques grâce aux racines de l'unité, implémenter le pivot de Gauss, découvrir que l'algorithme d'Euclide et le processsus d'orthonormalisation de Gram-Schmidt (deux algorithmes de niveau L2) sont en fait le *même algorithme !* (à une surcharge d'opérateur près)

Bases pédagogiques de la programmation

J'ai été récemment très inspiré par le cours d'initiation à la programmation de GdQuest: je pense qu'utiliser des outils comme un engine de jeu vidéo pour enseigner les bases de la programmation pourrait être une excellente idée: cela aiderait vraiment à la visualisation des concepts primitifs (boucle for, while etc…) à l'aide de personnage se déplaçant sur des grilles, ce qui rapelle les exercices de France IoI, qui sont malheureusement beaucoup moins accessibles (tout en ligne de commande). On peut même aller vers des algorithmes plus avancés (par exemple A^*). Je pense que cela peut-être aussi amusant/motivant pour les élèves de L1, si l'on propose les exercices sous forme de défi à accomplir.

L'exemple mentionné plus haut utilise gdscript et le Godot engine: il y aurais donc un coût cognitif supplémentaire risqué pour les élèves: ils vont devoir apprendre un langage (gdscript) et changer de langage très vite (dès le second semestre), mais je pense que cela peut aussi être bénéfique: un bon dévellopeur sait transposer les concepts d'un langage qu'il connait vers un autre. Une fois passé cette difficulté, j'espère que les élèves auront intégré que la plupart des languages impératifs se ressemblent, et qu'il faut plus s'intéresser aux concepts abstraits du language (présence d'objet, de fonction de première classe etc…) qu'à sa syntaxe.

Je serais intéréssé de dévelloper des TPs dans cet esprit à moyen terme, et de les tester sur le terrain.

Génie Logiciel

Je suis persuadé que le Génie Logiciel est un savoir faire, et qu'un savoir faire *s'apprend par la pratique*. Je pense notamment qu'il serait très formateur pour les étudiants de dévelloper des mini-projet sur une année (je dirais 3ème ou 4ème année post-bac) sous le format suivant:

- Les étudiants reçoivent en petit groupes une « demande de client » pour dévelloper une application: ils ont 2 TP pour faire un prototype.
- Au 3ème TP, et ensuite tout les 2 TP, le client « change d'avis »: il demande de nouvelles fonctionnalités, de changer des existantes etc… Les étudiants ont alors à nouveau 2 TP pour adapter leur projet à ses demandes.

On devrait surement fournir une partie du code (typiquement l'interface graphique) et une API avec laquelle ils peuvent interagir: cela permet de gagner du temps, et de les forcer à travailler dans un environement préexistant (ce qui est le cas en entreprise).

Je pense aussi qu'il est envisageable de recommencer à zéro à la mi-année, potentiellement sur un sujet « similaire » (client différent, même type d'application): cela permet aux étudiants un peu perdu de recommencer, et cela leur permet de regarder en arrière sur les mauvais choix d'architecture qu'ils ont fait et de ne pas les répéter.

Assistants de Preuves

Je serais aussi intéréssé par l'utilisation d'assistant de preuves dans les cours; des expériences ont déjà été effectuées avec plus ou moins de succès [43].

On peut espérer un double intêret

- L'enseignement dès la L1 peut potentiellement aider les élèves à comprendre comment bien rédiger des preuves formellement, ce qui leur serait utile en maths. On peut aussi essayer de le faire de manière ludique [44].
- Une fois les élèves familiarisé avec ces outils, on pourrait espèrer ouvrir des cours dans les années supérieures (3ème année post-bac et plus)
  - Dans des cursus plus théoriques voir la théorie derrière l'assitant, ce qui serait facilité par sa maitrise.

- Dans des cursus plus professionalisants (IUT, école d'ingénieurs), on pourrait aussi ouvrir des cours d'utilisation des assistants de preuves pour le logiciel: vérifier la correction de programmes, de manière automatique via des logiciels comme Frama-C, Creusot/Why3 [45] , directement avec des assistants comme Rocq [46], certifier qu'il n'y a pas de problèmes liés aux pointeurs dans des langages bas niveau via des logique de séparations etc…

5.2.4 - Bibliographie

[1] M. Kerjean, V. Maestracci, et M. Rogers, « Functorial Models of Differential Linear Logic », in *10th International Conference on Formal Structures for Computation and Deduction (FSCD 2025)*, M. Fernández, Éd., in Leibniz International Proceedings in Informatics (LIPIcs), vol. 337. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025, p. 26:1-26:17. doi: 10.4230/LIPIcs.FSCD.2025.26.

[2] V. Maestracci et P. Pistone, « The Lambda Calculus Is Quantifiable », in *33rd EACSL Annual Conference on Computer Science Logic (CSL 2025)*, J. Endrullis et S. Schmitz, Éd., in Leibniz International Proceedings in Informatics (LIPIcs), vol. 326. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025, p. 34:1-34:23. doi: 10.4230/LIPIcs.CSL.2025.34.

[3] B. Barras et V. Maestracci, « Implementation of Two Layers Type Theory in Dedukti and Application to Cubical Type Theory », in *LFMPT 2020 - Logical Frameworks and Meta-Languages: Theory and Practice 2020*, Paris, France, juin 2020. Consulté le: 27 février 2026. [En ligne]. Disponible sur: https://inria.hal.science/hal-03138145

[4] P.-L. Curien, « System L syntax for sequent calculi ».

[5] G. Geoffroy et P. Pistone, « A Partial Metric Semantics of Higher-Order Types and Approximate Program Transformations », in *CSL 2021 - Computer Science Logic*, Lubjana, Slovenia, janv. 2021. doi: 10.4230/LIPIcs.CSL.2021.23.

[6] T. Seiller, « Interaction graphs: Multiplicatives », *Annals of Pure and Applied Logic*, vol. 163, nº 12, p. 1808-1837, déc. 2012, doi: 10.1016/j.apal.2012.04.005.

[7] T. Seiller, « Zeta Functions and the (Linear) Logic of Markov Processes », *Logical Methods in Computer Science*, p. 10303, août 2024, doi: 10.46298/lmcs-20(3:18)2024.

[8] V. Maestracci, « Un petit pas vers une Logique Open Source », PhD Thesis, 2025.

[9] T. Seiller, « Interaction graphs: Additives », *Annals of Pure and Applied Logic*, vol. 167, nº 2, p. 95-154, févr. 2016, doi: 10.1016/j.apal.2015.10.001.

[10] T. Seiller, « Logique dans le facteur hyperfini : Géométrie de l' interaction et complexité », These de doctorat, 2012. Consulté le: 8 août 2025. [En ligne]. Disponible sur: https://theses.fr/2012AIXM4064

[11] T. Seiller, « Interaction Graphs: Full Linear Logic », in *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, in LICS '16. New York, NY, USA: Association for Computing Machinery, juill. 2016, p. 427-436. doi: 10.1145/2933575.2934568.

[12] M. Hamano, « A MALL geometry of interaction based on indexed linear logic », *Mathematical Structures in Computer Science*, vol. 30, nº 10, p. 1025-1053, nov. 2020, doi: 10.1017/S0960129521000062.

[13] V. Maestracci et T. Seiller, « From Geometry to Interaction ».

[14] B. Eng, « An exegesis of transcendental syntax : A journey into the logical machinery », These de doctorat, 2023. Consulté le: 18 août 2025. [En ligne]. Disponible sur: https://theses.fr/2023PA131018

[15] M. Bagnol, « On the Resolution Semiring ».

[16] T. Seiller, « Interaction graphs: Graphings », *Annals of Pure and Applied Logic*, vol. 168, nº 2, p. 278-320, févr. 2017, doi: 10.1016/j.apal.2016.10.007.

[17] S. Abramsky et R. Jagadeesan, « Games and Full Completeness for Multiplicative Linear Logic ». Consulté le: 27 février 2026. [En ligne]. Disponible sur: http://arxiv.org/abs/1311.6057

[18] D. Baelde, S. Delaune, C. Jacomme, A. Koutsos, et J. Lallemand, « The Squirrel Prover and its Logic », *ACM SIGLOG News*, vol. 11, nº 2, avr. 2024, doi: 10.1145/3665453.3665461.

[19] G. Bana et H. Comon-Lundh, « A Computationally Complete Symbolic Attacker for Equivalence Properties », in *2014 ACM SIGSAC Conference on Computer and Communications Security*, Scottsdale, United States: ACM, nov. 2014, p. 609-620. doi: 10.1145/2660267.2660276.

[20] B. Barak, R. Crubillé, et U. D. Lago, « On Higher-Order Cryptography (Long Version) ». Consulté le: 3 novembre 2025. [En ligne]. Disponible sur: http://arxiv.org/abs/2002.07218

[21] S. Abramsky et P.-A. Mellies, « Concurrent games and full completeness », in *Proceedings. 14th Symposium on Logic in Computer Science (Cat. No. PR00158)*, juill. 1999, p. 431-442. doi: 10.1109/LICS.1999.782638.

[22] F. Jafarrahmani, « Fixpoints of types in linear logic from a Curry-Howard-Lambek perspective », Theses, 2023. Consulté le: 27 février 2026. [En ligne]. Disponible sur: https://theses.hal.science/tel-04523738

[23] T. Seiller, « Mathematical Informatics », thesis, 2024. Consulté le: 17 mars 2025. [En ligne]. Disponible sur: https://theses.hal.science/tel-04616661

[24] C. Retoré, « Pomset logic: a logical and grammatical alternative to the Lambek calculus ».

[25] M. Vákár, « Syntax and Semantics of Linear Dependent Types ». Consulté le: 27 février 2026. [En ligne]. Disponible sur: http://arxiv.org/abs/1405.0033

[26] Z. Luo et Y. Zhang, « A Linear Dependent Type Theory ».

[27] P. Fu, K. Kishida, et P. Selinger, « Linear Dependent Type Theory for Quantum Programming Languages », *Logical Methods in Computer Science*, p. 6930, sept. 2022, doi: 10.46298/lmcs-18(3:28)2022.

[28] N. R. Krishnaswami, P. Pradic, et N. Benton, « Integrating Linear and Dependent Types », in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Mumbai India: ACM, janv. 2015, p. 17-30. doi: 10.1145/2676726.2676969.

[29] C. McBride, « I Got Plenty o' Nuttin' », *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. Springer International Publishing, Cham, p. 207-233, 2016. doi: 10.1007/978-3-319-30936-1_12.

[30] P. Clairambault, « Causal Investigations in Interactive Semantics », Habilitation à diriger des recherches, 2024. Consulté le: 27 février 2026. [En ligne]. Disponible sur: https://hal.science/tel-04523273

[31] P.-L. Curien et C. Faggian, « L-Nets, Strategies and Proof-Nets », *Computer Science Logic*, vol. 3634. Springer Berlin Heidelberg, Berlin, Heidelberg, p. 167-183, 2005. doi: 10.1007/11538363_13.

[32] A. Miquel, « Implicative algebras: a new foundation for realizability and forcing », *Mathematical Structures in Computer Science*, vol. 30, nº 5, p. 458-510, mai 2020, doi: 10.1017/S0960129520000079.

[33] A. Lucquin, L. Pellissier, et T. Seiller, « Linear Realisability and Implicative Algebras ». Consulté le: 13 mars 2026. [En ligne]. Disponible sur: http://arxiv.org/abs/2602.06576

[34] L. Cohen, É. Miquey, et R. Tate, « Evidenced Frames: A Unifying Framework Broadening Realizability Models », in *LICS 2021*, Rome, Italy, juill. 2021. doi: 10.1109/LICS52264.2021.9470514.

[35] P. Pistone, « On Proofs and Types in Second Order Logic ».

[36] J.-Y. Girard, « Transcendental syntax III: equality ».

[37] J. Koleilat, « A Tangent on Categorical Models of Differential Linear Logic ».

[38] C. Retoré, « Pomset logic: the other approach to non commutativity in logic », vol. 20. p. 299-345, 2021. doi: 10.1007/978-3-030-66545-6_9.

[39] P. de Groote, « Lambek Categorial Grammars as Abstract Categorial Grammars ».

[40] T. Ehrhard et L. Regnier, « The differential lambda-calculus », *Theoretical Computer Science*, vol. 309, nº 1, p. 1-41, déc. 2003, doi: 10.1016/S0304-3975(03)00392-X.

[41] P. Cousot, « Types as abstract interpretations », in *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '97*, Paris, France: ACM Press, 1997, p. 316-331. doi: 10.1145/263699.263744.

[42] J.-Y. Girard, « Wo, clef de voûte logique ».

[43] M. Kerjean *et al.*, « Utilisation des assistants de preuves pour l'enseignement en L1 », *La Gazette de la Société mathématique de France*, vol. 174, oct. 2022, Consulté le: 19 mars 2026. [En ligne]. Disponible sur: https://hal.science/hal-03979238

[44] « Lean Game Server ». Consulté le: 19 mars 2026. [En ligne]. Disponible sur: https://adam.math.hhu.de/

[45] « Program safely in Rust with the Why3 proof assistant | Inria ». Consulté le: 19 mars 2026. [En ligne]. Disponible sur: https://www.inria.fr/en/digital-security-rust-proof-assistant-why3

[46] « Software Foundations ». Consulté le: 19 mars 2026. [En ligne]. Disponible sur: https://softwarefoundations.cis.upenn.edu/

[47] C. Cohen, T. Coquand, S. Huber, et A. Mörtberg, « Cubical Type Theory: a constructive interpretation of the univalence axiom ». Consulté le: 27 février 2026. [En ligne]. Disponible sur: http://arxiv.org/abs/1611.02108

Annexe 1 : Undergrad

Cette section raconte un peu mon parcours de recherche, de Licence à Master, avant que je ne choisisse mon orientation « finale ».

Cela explique notamment mon intêret pour les assistants de preuves, ainsi que certaines des branches des maths que j'utilise parfois dans ma recherche.

1.1 - Cubical Agda

J'ai commencé à m'intéresser à ce domaine dès la L3. Mon premier stage de recherche était avec Thierry Coquand. A l'époque venait de sortir un nouvel assistant de preuve: Cubical Agda. Il est basé sur une théorie des types (un système logique) nommée Cubical Type Theory [47], qui est une forme de successeur de Homotopy Type Theory (HoTT).

L'intêret de HoTT est la présence d'un axiome, l'axiome d'Univalence, qui permet de transformer des isomorphismes en égalité. Pour l'expliquer simplement, cela permet de rendre rigoureux certains abus de langage courant en mathématiques comme « Le groupe à 2 élément ». En effet, si je prend le groupe $\mathbb{Z}/2\mathbb{Z}$, et que je renomme 0 en « zero », 1 en « un » et que je garde les mêmes lois, j'obtiens un groupe isomorphe, mais *techniquement différent* car en terme d'ensemble, $\{0, 1\} \neq \{\text{zéro}, \text{un}\}$. Dans HoTT, puisque ces deux groupes sont isomorphes ils sont égaux (noter que ce n'est donc pas l'égalité ensembliste), et on peut bien parler « du groupe » à 2 éléments.

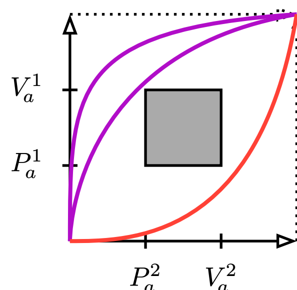
Le problème étant que l'Univalence est posée comme un axiome: une proposition/typage, mais HoTT n'a pas de programme pour la « justifier ». Notamment, une propriété importante de ces théories en général est qu'elles « calculent ». Si j'ai prouvé l'existence d'un nombre, je peut executer ma preuve (c'est un programme) et obtenir le dit nombre, mais ce n'est pas le cas de HoTT à cause de l'univalence. Cubical est une théorie qui à l'univalence et qui calcule. Le but de mon stage était d'implémenter des notions de théorie des catégories (complexe de chaine) dans cette assistant, notions qui nécessitaient la présence de calcul dans l'univalence pour être implémentée facilement.

1.2 - Topologie Algébrique Dirigée pour la Concurrence

J'ai ensuite effectué en M1 6 mois de stage sur la topologie algébrique dirigée pour l'informatique. On modélise des programmes parallèles avec sémaphore par un langage de programmation avec deux instruction P et V: P_a prend la ressource a, ce qui empêche les autres programmes en parallèle de la

prendre, et V_a la relache. Ainsi on ne peut pas avoir $P_a; P_a$ comme déroulé d'un programme, car le deuxième P_a essaie de prendre une ressource déjà prise.

Considérons deux programme simples et identiques, exécutés en parrallèle $P_a^1; V_a^1$ et $P_a^2; V_a^2$, ou l'on note P^1 ou P^2 pour se réferer à un programme, mais ce sont les mêmes instructions. On les représente comme les deux axes dans le dessin qui suit:



L'idée de ce domaine de recherche est de réaliser ces programmes comme des espaces topologiques: ici le carré en pointillé. On crée alors un trou dans cet espace, le carré gris au centre, pour représenter les zones « interdites », ici une situation ou les deux programmes auraient pris la ressource a en même temps.

On modélise alors les exécutions possible de ce programme comme des chemins *dirigés* dans cet espace topologique, allant de $(0, 0)$ en bas a gauche à $(3, 3)$ en haut à droite. Ces chemins sont intuitivement dirigés car il y a une fleche du temps: le programme ne peut pas revenir en arrière.

Sur l'exemple, le chemin rouge correspond à une execution de 2 suivi de 1, donc $P_a^2; V_a^2; P_a^1; V_a^1$, alors que les chemins violets l'inverse. En fait, on ne veut pas compter tout les chemins: les deux chemins violets correspondent à la même exécution, il faut donc une notion d'identification entre chemins dirigées, c'est *l'homotopie dirigée*. Le but de mon stage était de ma familariser avec ces concepts, et d'essayer de trouver des méthodes pour calculer syntaxiquement de telles classes d'équivalence de chemin.

Ces connaissances m'ont ultimement servie bien plus tard, dans ma thèse, pour produire un papier actuellement en soumission: [13].

Finalement, en master j'ai fini par revenir sur de la théorie des types et des assistants de preuves

1.3 - Théorie des Types et Dedukti

J'ai réalisé lors de mon stage de Master 2 sous la direction de Bruno Barras et Valentin Blot au sein du LSV (maintenant LMF) à l'ENS Paris-Saclay, dans la Deducteam (équipe fondée par Gilles Dowek) ce qui a donné lieu à une petite publication: Implementation of Two Layers Type Theory in Dedukti and Application to Cubical Type Theory [3]

Leur logiciel, Dedukti est une implémentation d'un Logical Framework, qui permet d'encoder en son sein différentes théories mathématiques (notamment celle de Rock, Agda etc…) le but final étant de traduire les preuves faites dans différents assistant de preuves, qui ne sont pas inter-opérables, et de les rassembler au sein de Logipédia, une sorte de bibliothèque en ligne. Plus généralement, pour n assistant de preuves qu'on veut faire opérer ensemble, on évite d'avoir n^2 traduction à faire en passant par ce « hub », on en fait que $2n$.

Le but ici était d'essayer d'obtenir des traduction de Cubical Agda, l'assistant de preuve que j'avais utilisé en L3. J'ai pour cela implémenté dans Dedukti 2LTT, une théorie des types « à 2 niveaux », ce qui permet d'éviter d'encoder les règles d'égalité de Cubical comme des réecritures dans Dedukti (ce

qui est a priori potentiellement difficile), en les encodant dans un deuxième niveau de syntaxe au lieu de les encoder « méta-théoriquement ».