

Synthèse TAS 2025

Meaning-Typed Programming & Formalisation de types pour ByLLM

Breton Noé

n°21516014

Boudrouss Réda

n°28712638



Sorbonne Université
France

Organisation du rapport

Le rapport est organisé en 3 parties :

— **Partie I : Synthèse de l'article MTP**

Synthèse de l'article "MTP : A Meaning-Typed Language Abstraction for AI-Integrated Programming"
(<https://dl.acm.org/doi/10.1145/3763092>)

— **Partie II – Contribution personnelle : esquisse de formalisation et vérification de types pour ByLLM**

Cette partie présente une formalisation abstraite du paradigme MTP indépendante du langage Jac, incluant des garanties de sûreté.

— **Partie III – Contribution personnelle : jacEMMA**

Cette partie décrit des exemples d'utilisation de byllm et une implémentation personnelle d'un système de prédiction de trajectoire pour véhicules autonomes utilisant ByLLM et Jac. Inspirée par LightEMMA et une version personnelle de LightEMMA produite durant un stage à l'université d'Aalto (supervisée par Azam Shoabib — article non encore disponible).

- Le principe de LightEMMA est repris avec un seul prompt. Mon travail en stage a démontré qu'un seul prompt offrait des performances comparables à plusieurs prompts pour certains modèles, notamment les modèles Cosmos de NVIDIA ; pour les modèles Qwen, elles sont légèrement inférieures, mais correctes.
- Le seul code repris de LightEMMA concerne certaines fonctions de `utils.py`, comme la conversion des driving intents en waypoints.

Toutes nos contributions peuvent être trouvées dans le dépôt GitHub suivant : <https://github.com/rboudrouss/TAS-ByLLM>

Vous trouverez aussi ici le lien vers un fork de JacLang contenant l'ajout d'une commande `jac tools ir semantic` qui permet de visualiser le registre sémantique généré par le MT-IR : <https://github.com/rboudrouss/jaseci>

Partie I

Synthèse de l'article

”MTP : A Meaning-Typed Language
Abstraction
for AI-Integrated Programming”

1. Introduction

1.1. Contexte

Avec l'avènement de l'intelligence artificielle générative et des Large Language Models (LLMs), la manière dont nous développons les logiciels évolue rapidement. Si de nombreux travaux se sont concentrés sur l'utilisation des LLMs pour la génération de code, une tendance émergente consiste à intégrer les LLMs comme composants fonctionnels essentiels qui s'exécutent au runtime. Ces applications AI-Integrated exploitent l'IA générative pendant l'exécution pour réaliser des fonctionnalités de plus en plus critiques du programme, fusionnant ainsi la programmation conventionnelle avec des capacités pilotées par l'IA.

La construction d'applications AI-Integrated reste cependant un défi complexe en raison des différences fondamentales entre les paradigmes de programmation. Dans la programmation conventionnelle, le code décrit des opérations précises effectuées sur des variables explicitement définies. En revanche, les LLMs traitent du texte en langage naturel en entrée et produisent des sorties textuelles. Pour intégrer des LLMs dans les programmes, les développeurs doivent donc manuellement construire des entrées textuelles, un processus connu sous le nom de prompt engineering.

Plusieurs frameworks tels que LangChain, LMQL, DSPy et SGLang ont été développés pour faciliter le prompt engineering. Bien qu'ils fournissent des outils plus sophistiqués, ils introduisent en pratique des couches supplémentaires de complexité. Les développeurs restent responsables de la construction manuelle des prompts, doivent se familiariser avec de nouveaux langages ou syntaxes spécialisées, et font face aux défis de conversion des sorties LLM en objets compatibles avec leur code.

1.2. Problématique et motivation

L'approche du prompt engineering impose aux développeurs plusieurs problématiques : la construction manuelle de prompts détaillés incluant descriptions du problème, spécifications d'entrée et règles de génération, la définition explicite et laborieuse des formats de sortie, particulièrement pour les objets profondément imbriqués, la conversion des sorties textuelles du LLM en objets typés compatibles avec le reste de l'application, nécessitant souvent de multiples fonctions auxiliaires et la gestion des erreurs et des mécanismes de retry pour pallier la nature non-déterministe des LLMs.

Dans cet article, les auteurs introduisent le Meaning-Typed Programming (MTP), une approche visant à simplifier la création d'applications AI-Integrated. Le paradigme MTP repose sur trois composants principaux : l'opérateur `by` qui permet l'intégration de fonctionnalités LLM dans le code, MT-IR (Meaning-Typed Intermediate Representation) qui capture la sémantique du code, et MT-Runtime qui automatise les interactions avec les LLMs. L'idée centrale de ce travail est que le code source contient déjà de l'information sémantique, car il est écrit pour être compris par des développeurs. Avec les capacités actuelles des LLMs, il devient possible d'inférer les intentions du code sans nécessiter de descriptions explicites ou de construction de prompts par les développeurs. Les noms de fonctions, les paramètres et les types portent du sens qui peut être exploité automatiquement pour générer les prompts, traduisant ainsi l'intention du code en instructions pour le LLM.

À titre d'illustration, considérons une fonction qui génère le prochain niveau d'un jeu vidéo. Avec les frameworks actuels, cette tâche nécessite la construction manuelle d'un prompt détaillé, la spécification explicite du format de sortie, et l'implémentation de mécanismes de conversion et de validation. En revanche, l'approche MTP permet d'exprimer cette même fonctionnalité en une seule ligne : `def get_next_level(prev_levels: list[Level]) -> Level by llm`. Cette déclaration offre plusieurs avantages : une abstraction du prompt engineering, l'élimination de la conversion manuelle des données, et un code plus maintenable. Cela soulève la question : comment MTP peut-il automatiser ce processus ?

2. Solution proposée : Meaning-Typed Programming

L'objectif de ce travail est d'exploiter les informations sémantiques présentes dans le code pour automatiser le prompt engineering. Plutôt que d'exiger des développeurs qu'ils construisent manuellement des prompts

détaillés ou des annotations, l'approche proposée utilise la structure et la sémantique du code pour générer des entrées pour les LLMs.

2.1. Objectifs de conception

Pour atteindre cet objectif, les auteurs identifient trois axes de travail :

O1 - Abstraction de langage

Fournir une interface de programmation simple et flexible qui cache la complexité du prompt engineering aux développeurs.

O2 - Extraction de sémantique

Développer des mécanismes d'extraction des informations sémantiques du code nécessaires à la génération automatique de prompts.

O3 - Système d'exécution

Mettre en place un système capable de combiner sémantique statique et valeurs dynamiques pour orchestrer la génération de prompts et l'interprétation des résultats.

2.2. Défis techniques

Ces objectifs posent quatre défis techniques :

C1 - Simplicité et flexibilité de l'abstraction

L'abstraction doit supporter différentes méthodes d'intégration LLM tout en restant accessible aux développeurs. La difficulté est de trouver un équilibre entre une interface simple qui réduit la courbe d'apprentissage et une flexibilité suffisante pour couvrir divers cas d'usage à différents endroits du code.

C2 - Sélection des informations sémantiques

Inclure tout le code source dans les prompts serait coûteux en tokens, il faut donc identifier et extraire uniquement les informations sémantiques pertinentes. Cette tâche est complexifiée par la dispersion de ces informations dans plusieurs fichiers, ce qui requiert une analyse de la codebase.

C3 - Accès au contexte d'exécution

Le système doit accéder aux valeurs des variables à l'exécution pour les combiner avec la sémantique statique. Si cet accès est direct pour les paramètres de fonctions, il est plus complexe pour les méthodes d'objets qui nécessitent également l'accès aux attributs de l'instance.

C4 - Robustesse face aux sorties LLM

Les LLMs produisent des sorties non-déterministes qui peuvent dévier des formats attendus. Le système doit parser ces sorties, gérer les erreurs et implémenter des mécanismes de retry lorsque nécessaire.

2.3. Architecture de la solution

Pour répondre à ces objectifs, le paradigme Meaning-Typed Programming (MTP) repose sur trois composants :

1. **L'opérateur by** : une abstraction au niveau du langage Jac qui permet l'intégration de fonctionnalités LLM (répond à **O1**).
2. **MT-IR (Meaning-Typed Intermediate Representation)** : une représentation intermédiaire générée lors de la compilation qui capture les informations sémantiques du code (répond à **O2**).
3. **MT-Runtime** : un moteur d'exécution qui gère l'intégration avec les LLMs, combinant les informations sémantiques avec les valeurs au runtime pour générer les prompts et interpréter les sorties (répond à **O3**).

Le système MTP fonctionne selon un pipeline en deux phases. Lors de la compilation, MT-IR extrait les informations sémantiques du code source. À l'exécution, MT-Runtime utilise ces informations combinées aux valeurs dynamiques pour générer les prompts, invoquer le LLM, et interpréter les résultats.

3. Architecture et composants de MTP

3.1. L'opérateur by : abstraction au niveau du langage

Pour répondre au défi C1, les auteurs introduisent l'opérateur `by` comme abstraction centrale de MTP. Sa syntaxe de base est la suivante :

```
<code construct> by llm_ref(model_hyperparameters)
```

où `llm_ref` est une référence à un modèle de langage et `model_hyperparameters` sont des paramètres optionnels pour configurer le comportement du modèle. Cette syntaxe permet de réaliser la tâche définie à gauche de `by` en utilisant le modèle de langage spécifié à droite.

L'opérateur `by` peut être utilisé dans trois contextes différents : les définitions de fonctions, l'initialisation d'objets et les définitions de méthodes de classe. Dans la suite de ce chapitre, M représente l'ensemble des modèles de langage disponibles, et `eval(\cdot)` désigne la fonction d'évaluation sémantique.

3.1.1. by dans les définitions de fonctions

Considérons une fonction f avec des paramètres p_1, p_2, \dots, p_n de types T_1, T_2, \dots, T_n respectivement, et un type de retour T_r . En utilisant l'opérateur `by`, cette fonction peut être intégrée avec un LLM $m \in M$ ayant des hyperparamètres θ comme suit :

```
def f(p1: T1, ..., pn: Tn) -> Tr by m(theta)
```

À l'exécution, l'invocation de cette fonction avec les arguments v_1, \dots, v_n pour retourner v_r de type T_r est sémantiquement équivalente à :

```
vr = eval(f(v1, ..., vn))
= invoke-model(m, theta, f, [v1, ..., vn], [(T1, ..., Tn), Tr])
```

Le modèle m reçoit la signature de la fonction f , les arguments effectifs (v_1, \dots, v_n) avec leurs types correspondants (T_1, \dots, T_n) , et le type de retour attendu T_r . En exploitant ce contexte sémantique, le modèle peut raisonner sur le comportement attendu de la fonction et générer une sortie appropriée conforme à T_r .

3.1.2. by dans l'initialisation d'objets

L'opérateur `by` peut également être utilisé lors de l'initialisation d'objets pour permettre à un LLM de compléter les valeurs manquantes. Considérons une classe C avec des attributs a_1, \dots, a_n de types T_1, \dots, T_n . Si seuls les k premiers attributs sont fournis lors de l'initialisation ($k < n$), les attributs restants peuvent être complétés par $m \in M$ avec des hyperparamètres θ :

```
C(a1, ..., ak) by m(theta)
```

À l'exécution, lors de la création d'un tel objet, le système appelle le modèle avec le nom de la classe C , les valeurs des attributs fournis (v_1, \dots, v_k) , leurs types (T_1, \dots, T_k) et les types des attributs restants (T_{k+1}, \dots, T_n) . Le modèle utilise ces informations pour générer des valeurs appropriées pour les attributs manquants afin d'instancier complètement un objet obj de type C :

```
obj = eval(C(v1, ..., vk) by m(theta))
= invoke-model(m, theta, C, [v1, ..., vk], [(T1, ..., Tk), (Tk+1, ..., Tn)])
```

3.1.3. by dans les méthodes de classe

L'opérateur `by` peut aussi être utilisé pour définir des méthodes au sein d'une classe. Considérons une méthode mth au sein d'une classe C , qui prend des paramètres p_1, \dots, p_n de types T_1, \dots, T_n , et retourne une valeur de type T_r . Cette méthode peut être implémentée en utilisant $m \in M$, configuré avec des hyperparamètres θ :

```
class C:
    def mth(p1: T1, ..., pn: Tn) -> Tr by m(theta)
```

À l'exécution, lorsque cette méthode est appelée sur une instance d'objet `obj` avec les valeurs d'entrée v_1, \dots, v_n , l'évaluation pour retourner v_r de type T_r procède comme suit :

```
vr = eval(obj.mth(v1, ..., vn))
= invoke-model(m, theta, C.mth, obj, [v1, ..., vn], [C, (T1, ..., Tn), Tr])
```

Contrairement aux appels de fonctions autonomes, les méthodes de classe ont accès à l'état interne de l'instance. Lors de l'évaluation, le modèle reçoit la signature de la méthode `C.mth`, l'instance d'objet `obj`, les valeurs d'entrée avec leurs types, et le type de retour attendu. L'objet inclut tous les attributs de la classe avec leurs valeurs courantes et leurs informations de type, permettant au modèle d'utiliser ce contexte dans son raisonnement.

3.1.4. Garanties comportementales

L'opérateur `by` offre deux garanties importantes grâce à MT-IR et MT-Runtime :

Sûreté de type : Le système vérifie systématiquement que la sortie générée par le LLM correspond bien au type T_r attendu. Si la conversion échoue, une erreur de type est levée.

Comportement temporel : Le moment d'exécution de l'opérateur `by` dépend du contexte d'utilisation. Pour les fonctions et méthodes, l'exécution se produit uniquement lorsque la fonction est appelée. En revanche, pour l'initialisation d'objets, l'opérateur `by` s'exécute directement au moment de l'instanciation de l'objet.

3.2 MT-IR (Meaning-Typed Intermediate Representation)

À chaque site d'appel `by`, le système doit effectuer une analyse pour extraire les informations sémantiques du code nécessaires à la génération de prompts. Ces informations (types, signatures de fonctions) existent dans le code source mais risquent d'être perdues lors de la compilation en bytecode. Il faut donc les capturer avant cette transformation.

Les auteurs proposent MT-IR, une représentation intermédiaire qui stocke ces informations sémantiques. Comme une Intermediate Representation classique facilite la génération de code machine, MT-IR facilite la génération de prompts. À l'exécution, MT-Runtime utilise MT-IR sans accéder au code source original.

La construction de MT-IR se déroule en plusieurs phases :

3.2.1 Génération du registre sémantique

Lors de la compilation, le système construit un registre sémantique à partir de l'AST de chaque module. Ce registre centralise les noms de variables, fonctions, classes et leurs types. Contrairement à une table des symboles classique qui stocke uniquement les définitions, ce registre inclut aussi les usages et établit des liens entre eux.

3.2.2 Extraction sémantique par site d'appel `by`

Pour chaque site d'appel `by`, le système extrait d'abord les informations directes : nom de fonction/méthode/classe, types des paramètres, type de retour, modèle et hyperparamètres (lignes 3-9 de l'algorithme 1). Ces éléments correspondent aux paramètres passés à MT-Runtime selon l'équation donnée plus tôt : $(f, [v_1, \dots, v_n], [(T_1, \dots, T_n), T_r], m, \theta)$.

Ensuite, le système vérifie si les types sont primitifs (int, float, str). Pour les types non-primitifs, il parcourt récursivement le registre pour extraire leurs définitions complètes. Par exemple, un type `Level` peut dépendre de `Map`, qui lui-même dépend de `Wall` et `Position`. Les liens usage-définition du registre permettent cette traversée jusqu'aux types primitifs. Cette résolution récursive, implémentée par la fonction `ExtractTypeDefinition`, permet de capturer toute la hiérarchie de types nécessaire et répond au défi **C2**.

3.2.3 Implémentation dans le langage Jac

MTP est implémenté dans Jac, un sur-ensemble de Python distribué comme package PyPI. Jac fournit une commande CLI `jac` qui remplace le compilateur Python standard en ajoutant une phase de compilation personnalisée avant de générer du bytecode Python standard.

Le processus de compilation MTP dans Jac comporte six étapes. D'abord, un passage de compilation génère MT-IR en extrayant les éléments sémantiques de l'AST. Ces informations sont affinées par plusieurs analyses. L'AST Jac est ensuite transformé en AST Python, où les constructions `by` sont remplacées par des appels MT-Runtime. Une vérification de types MyPy enrichit ensuite MT-IR avec des informations de types supplémentaires. MT-IR est enregistré dans la bibliothèque MT-Runtime pour être disponible à l'exécution. Enfin, le bytecode Python est généré, préservant dans MT-IR les informations sémantiques qui auraient été perdues dans une compilation classique.

3.3 MT-Runtime : moteur d'exécution automatisé

Les programmes AI-Integrated effectuent l'inférence LLM pendant l'exécution, car le prompt nécessite à la fois les informations sémantiques et les valeurs dynamiques disponibles au moment de l'appel `by`. Cela requiert un système capable de fusionner MT-IR avec les valeurs des variables à l'exécution pour produire le prompt final. Les auteurs proposent MT-Runtime, un moteur intégré à la machine virtuelle python qui se déclenche automatiquement lors des appels `by`. Ce composant répond à l'objectif **O3** et traite les défis **C3** et **C4**.

3.3.1 Synthèse de prompts

À chaque appel `by`, MT-Runtime récupère la portion de MT-IR associée à ce site d'appel et l'utilise pour instancier un template de prompt. Le système construit alors un prompt qui combine la sémantique statique, les valeurs des variables pertinentes, et le type de retour attendu.

Puisque MT-Runtime s'exécute dans la machine virtuelle, il accède directement aux valeurs des variables. La sémantique provient de MT-IR tandis que les valeurs sont extraites du graphe d'objets du langage. Le prompt résultant structure ces informations de manière spécifique : le nom de la fonction et les éventuels commentaires définit l'action à réaliser, les types et leurs définitions imbriquées sont regroupés dans une section dédiée, et le schéma de sortie attendu est explicitement spécifié.

3.3.2 Conversion vers le type attendu

Une fois la réponse du LLM obtenue, MT-Runtime doit la transformer en une variable du type attendu. Cette étape traite le défi **C4**.

Le prompt ayant déjà demandé au modèle de produire une sortie conforme au schéma d'objet Python, MT-Runtime utilise `ast.literal_eval()` pour évaluer directement cette sortie. Par exemple, pour une variable de type `Person` avec les attributs `name: str` et `dob: str`, le LLM génère `Person(name="Albert Einstein", dob="03/14/1879")`, qui s'évalue immédiatement en un objet `Person` valide.

Si la conversion échoue, MT-Runtime construit un nouveau prompt qui signale l'erreur au modèle. Ce prompt de correction indique la sortie incorrecte et rappelle le type attendu. Le système réitère ce processus jusqu'à obtenir une sortie valide ou atteindre le nombre maximal de tentatives configuré par le développeur. Dans ce dernier cas, une exception de type est levée.

4. Implémentation et évaluation

4.1. Implémentation dans Jac

MTP est implémenté dans Jac, un sur-ensemble de Python développé par Jaseci Labs et distribué comme package PyPI. Jac étend Python avec de nouvelles primitives de langage tout en restant compatible avec l'écosystème Python existant. L'opérateur `by` constitue une primitive native de Jac, tandis que MT-IR et MT-Runtime sont intégrés respectivement comme passe de compilation et plugin d'exécution.

Le pipeline de compilation Jac transforme le code source en plusieurs étapes. L'AST Jac est d'abord généré, puis transformé en AST Python enrichi d'annotations de types. Durant cette transformation, une passe de compilation dédiée construit MT-IR en extrayant les informations sémantiques. Le bytecode Python standard est ensuite produit, avec MT-IR stocké séparément pour être accessible à MT-Runtime lors de l'exécution dans la machine virtuelle Python.

4.2. Évaluation

Les auteurs évaluent MTP selon quatre axes : la réduction de complexité du code, la précision des programmes, l'efficacité en termes de tokens et de temps d'exécution, et la robustesse face aux pratiques de codage. L'évaluation s'appuie sur une suite de 13 benchmarks couvrant diverses tâches (résolution de problèmes mathématiques, traduction, génération de contenu, raisonnement) et compare MTP aux frameworks LMQL et DSPy.

4.2.1. Réduction de la complexité du code

Les auteurs mesurent le nombre de lignes de code modifiées pour intégrer une fonctionnalité LLM dans un programme existant. Sur l'ensemble des benchmarks, MTP nécessite entre $2.3\times$ et $7.5\times$ moins de lignes que LMQL, et entre $1.3\times$ et $10.7\times$ moins que DSPy. Cette réduction s'explique par l'abstraction offerte par l'opérateur `by` qui élimine la construction manuelle de prompts requise par LMQL et les annotations de types détaillées imposées par DSPy.

Une étude utilisateur menée auprès de 20 développeurs confirme ces résultats quantitatifs. Les participants ont implémenté trois tâches de complexité croissante avec chaque framework. MTP obtient les meilleurs taux de réussite sur deux des trois tâches et montre la performance la plus stable. Les retours qualitatifs soulignent la simplicité de MTP, plusieurs participants notant que “le code MTP est plus court” et que “la conversion automatique entre types de données se fait facilement”.

4.2.2. Précision des programmes

L'évaluation de la précision porte sur les 13 benchmarks avec GPT-4o, chacun exécuté 100 fois. La précision moyenne atteint 90.85% pour LMQL, 98.23% pour DSPy et 98.92% pour MTP. MTP surpassé DSPy sur 9 des 13 benchmarks, démontrant que l'automatisation du prompt engineering ne dégrade pas la qualité des résultats.

Sur le benchmark GSM8K (300 problèmes mathématiques), les auteurs observent une tendance intéressante à travers l'évolution des modèles. Avec les modèles récents (GPT-4o, Llama 3.1), la précision de MTP s'améliore continuellement tandis que celle de DSPy stagne voire régresse légèrement. Cette observation suggère que les modèles plus performants comprennent mieux la sémantique du code, réduisant le besoin de prompt engineering complexe. MTP atteint ainsi près de 90% de précision sur GPT-4o, surpassant même DSPy en mode compilé qui nécessite des exemples d'entraînement supplémentaires.

4.2.3. Efficacité en tokens et temps d'exécution

MTP consomme systématiquement moins de tokens que DSPy sur tous les benchmarks. Cette réduction se traduit directement par des économies de coûts d'API allant jusqu'à $4.5\times$ sur certains benchmarks. Les prompts générés par MT-Runtime sont plus concis car MT-IR extrait uniquement les informations sémantiques pertinentes, évitant la verbosité des prompts DSPy.

Le temps d'exécution suit la même tendance. MTP obtient des accélérations allant jusqu'à $4.75\times$ par rapport à DSPy sur certains benchmarks, grâce à la réduction du nombre de tokens à traiter par le LLM. L'overhead introduit par MT-Runtime reste minimal car la synthèse de prompts s'effectue de manière efficace.

4.2.4. Robustesse aux pratiques de codage

Les auteurs testent la sensibilité de MTP à la dégradation de la qualité du code en renommant progressivement les identifiants du programme de génération de niveaux. Avec 25% et 50% d'identifiants renommés de manière

abrégée, MTP maintient 98% de précision. À 75% de renommage, la précision chute à 70%, et à 100% elle atteint 20%. Ces résultats montrent que MTP tolère une dégradation modérée de la qualité du code, mais que la sémantique reste essentielle au bon fonctionnement du système.

5. Discussion

5.1. Limitations sémantiques et dépendance à la qualité du code

L'approche MTP repose sur l'hypothèse que le code source contient suffisamment d'informations sémantiques pour générer automatiquement des prompts pertinents. Les résultats empiriques confirment cette hypothèse dans une certaine mesure : la réduction de $2.3 \times$ à $10.7 \times$ du nombre de lignes de code et la précision moyenne de 98.92% démontrent l'efficacité de l'extraction automatique.

Cependant, MTP fait face à deux limitations fondamentales. Premièrement, le système s'appuie exclusivement sur les informations syntaxiques du code (noms de variables, signatures de fonctions, annotations de types) pour inférer l'intention du développeur. De nombreuses contraintes métier ne peuvent être exprimées par cette seule structure : “le niveau de difficulté doit augmenter progressivement mais pas de plus de 20%”, “la position du joueur ne doit jamais coïncider avec un mur”, ou “générer des ennemis plus agressifs en mode difficile” nécessitent des descriptions explicites que MTP ne peut extraire automatiquement. L'approche risque ainsi de produire des résultats technique corrects (conformes aux types) mais sémantiquement inadéquats (ne respectant pas les contraintes métier implicites).

Deuxièmement, l'expérience de dégradation révèle une dépendance à la qualité du nommage : à 75% d'identifiants renommés, la précision chute à 70%. Cette sensibilité soulève des questions sur l'applicabilité de MTP aux codebases legacy ou aux projets avec des pratiques de nommage inconsistentes, bien que la robustesse jusqu'à 50% de dégradation suggère une tolérance acceptable pour la plupart des contextes pratiques.

5.2. Typage graduel de Python versus typage fort

Le système de types constitue le mécanisme principal d'extraction sémantique dans MTP. MT-IR exploite les annotations de types pour construire une représentation complète des structures de données, résolvant récursivement les dépendances (par exemple, `Level` → `Map` → `Wall`, `Position`). Cette approche transforme les annotations de types en ressource pour la génération de prompts.

L'implémentation actuelle dans Jac/Python souffre cependant des limitations inhérentes au typage graduel. Python permet l'utilisation de types génériques (`Any`, `dict`, `list`) qui offrent peu d'informations sémantiques exploitables. De plus, les annotations de types ne sont pas vérifiées strictement à la compilation : elles peuvent être absentes, incorrectes ou incomplètes sans que le compilateur ne signale d'erreur. Cette permissivité réduit la fiabilité de MT-IR. L'article ne documente pas non plus la gestion des constructions avancées (unions de types `Union[int, str]`, génériques avec paramètres, types récursifs).

L'implémentation de MTP dans un langage à typage statique fort (TypeScript, Rust, Haskell) présenterait plusieurs avantages significatifs. Les systèmes de types algébriques permettent d'exprimer des contraintes structurelles complexes : un type `Option<T>` en Rust encode explicitement la possibilité d'absence de valeur, tandis qu'en Python cette information est implicite. Les types raffinés pourraient exprimer des contraintes comme “un entier strictement positif”, enrichissant les prompts générés. Un système de types vérifié statiquement garantirait que les annotations sont cohérentes avec l'utilisation effective des variables, éliminant une source d'erreur potentielle. Les interfaces TypeScript ou les traits Rust permettraient de communiquer au LLM non seulement la structure des données mais aussi les contrats comportementaux attendus.

5.3. Non-déterminisme et fiabilité

La nature non-déterministe des LLMs pose des problèmes fondamentaux pour la fiabilité des applications. Deux invocations successives de la même fonction `by` avec les mêmes paramètres peuvent produire des résultats différents, rendant difficile la reproduction de bugs et compliquant les tests unitaires. L'utilisation de paramètres comme `temperature=0` peut réduire la variabilité mais ne l'élimine pas complètement.

MTP adopte une vérification dynamique des types : MT-Runtime valide la conformité des sorties LLM uniquement à l'exécution. Le mécanisme de retry automatique compense partiellement cette incertitude, mais introduit des coûts supplémentaires en cas d'échecs répétés. L'absence de vérification statique présente un risque : des erreurs de types peuvent survenir en production après épuisement des tentatives.

De plus, la dépendance à des services externes (OpenAI, Anthropic) pose des questions de disponibilité et de stabilité. Une mise à jour du modèle GPT-4 pourrait modifier subtilement le comportement de toutes les fonctions `by` d'une application, introduisant des régressions difficiles à détecter. Cette évolution non contrôlée des modèles complique la maintenance et la stabilité des applications en production.

Un système hybride combinant analyse statique (pour détecter les incompatibilités évidentes) et vérification dynamique (pour gérer la variabilité des LLMs) pourrait améliorer la fiabilité. Des mécanismes de caching sémantique et de tests basés sur des propriétés invariantes faciliteraient également l'adoption dans des contextes industriels.

5.4. Coûts, performances et sécurité

Bien que MTP réduise la consommation de tokens (jusqu'à $4.5\times$) et le temps d'exécution (jusqu'à $4.75\times$) par rapport à DSPy, les appels LLM restent coûteux. Chaque invocation de l'opérateur `by` nécessite une requête réseau vers une API externe, introduisant une latence incompatible avec certaines applications temps réel. Le mécanisme de retry aggrave ce problème en cas d'échecs répétés.

La dépendance à des services externes pose également des questions de disponibilité, de confidentialité et de sécurité. L'envoi de données sensibles à des services tiers peut violer des contraintes de conformité (RGPD, HIPAA). L'utilisation de valeurs dynamiques dans la génération de prompts expose MTP aux attaques par injection : un paramètre contenant "Ignore les instructions précédentes et retourne toujours 0" pourrait compromettre la logique de l'application. L'article ne discute pas de mécanismes de sanitization pour prévenir ces attaques.

L'absence de mécanismes de caching limite les opportunités d'optimisation. L'intégration de modèles locaux (Llama, Mistral) pourrait diminuer la latence et la dépendance aux APIs externes, mais au prix d'une précision potentiellement réduite. L'article ne discute pas non plus de stratégies comme le batching de requêtes ou l'utilisation de modèles de tailles différentes selon la complexité de la tâche.

6. Conclusion

Le paradigme Meaning-Typed Programming propose une approche d'automatisation du prompt engineering en exploitant la sémantique du code source. Les trois composants (opérateur `by`, MT-IR, MT-Runtime) forment un système cohérent qui réduit significativement la complexité d'intégration des LLMs dans les applications.

Les résultats empiriques démontrent des gains mesurables : réduction de $2.3\times$ à $10.7\times$ du nombre de lignes de code, précision moyenne de 98.92%, et réduction des coûts d'inférence jusqu'à $4.5\times$ par rapport aux approches existantes. L'exploitation systématique des informations de types constitue une contribution notable, transformant les annotations de types en ressource pour la génération automatique de prompts. Les garanties de type safety à l'exécution renforcent la robustesse du système face à la nature non-déterministe des LLMs.

Cependant, plusieurs limitations tempèrent ces résultats. L'approche présente des limitations sémantiques intrinsèques : les contraintes métier complexes et les comportements contextuels ne peuvent être capturés par la seule structure syntaxique du code, risquant de produire des résultats techniquement corrects mais sémantiquement inadéquats. L'implémentation dans Jac/Python souffre des limitations du typage graduel (types génériques `Any`, `dict`, `list` peu informatifs, absence de vérification stricte à la compilation), alors qu'un langage à typage statique fort (TypeScript, Rust, Haskell) permettrait d'exprimer des contraintes structurelles complexes via les types algébriques et de garantir la cohérence des annotations. L'implémentation limitée au langage Jac, peu répandu, constitue également une barrière majeure à l'adoption.

Le non-déterminisme des LLMs pose des problèmes fondamentaux pour la fiabilité (reproduction difficile des bugs, tests unitaires complexifiés, risques d'erreurs en production). La dépendance à des services externes évolutifs complique la maintenance, une mise à jour de modèle pouvant introduire des régressions subtiles. Les questions de sécurité (injection de prompts, confidentialité des données) et de coûts (latence réseau, absence de caching) limitent l'applicabilité à des systèmes critiques ou temps réel.

Ces limitations ouvrent plusieurs perspectives d'amélioration. L'extension à des langages fortement typés (TypeScript, Rust, Haskell) enrichirait l'extraction sémantique et renforcerait les garanties de correction grâce aux types algébriques et à la vérification statique. Un système hybride combinant analyse statique et vérification dynamique améliorerait la fiabilité tout en préservant la flexibilité face à la variabilité des LLMs. L'intégration de mécanismes de caching sémantique, de tests basés sur des propriétés invariantes, et de support pour des modèles locaux réduirait les coûts et la dépendance aux services externes. Enfin, des mécanismes de sanitization des entrées préviendraient les attaques par injection de prompts, renforçant la sécurité du système.

MTP représente une contribution significative au domaine de la programmation intégrée à l'IA, démontrant la viabilité de l'exploitation de la sémantique du code pour l'automatisation du prompt engineering. L'approche ouvre des pistes de recherche prometteuses sur l'interaction entre systèmes de types et modèles de langage, tout en révélant les limites fondamentales d'une approche purement syntaxique face à la complexité sémantique des applications réelles.

Partie II

Esquisse de formalisation et vérification de types pour ByLLM

1. Motivation pour la formalisation

L'implémentation de MTP dans Jac est une contribution concrète, mais elle est limitée à un seul langage. Pour généraliser la solution, il est nécessaire de la formaliser de manière indépendante du langage. Cela permettra de raisonner sur le paradigme MTP de manière abstraite et de le porter à d'autres langages.

De plus, byLLM implémente plusieurs vérifications au runtime : la génération de schémas JSON lève une exception si le type n'est pas supporté en entrée et la validation pydantic lève une exception si le type de sortie n'est pas respecté. Sauf que ces vérifications sont effectuées à l'exécution, ce qui n'est pas idéal. Dans cette deuxième partie, nous proposons une esquisse de formalisation du paradigme MTP, incluant des garanties de sûreté de type statique.

2. Formalisation du système MTP

2.1. Syntaxe

La syntaxe de MTP est définie par l'ensemble des expressions suivantes :

$$\text{Types } \tau ::= B \mid \tau_1 \rightarrow \tau_2 \mid \text{List}\langle\tau\rangle \mid \text{Record}\{\ell_i : \tau_i\}_{i \in I}$$

où B représente les types de (int, str, bool, float).

$$\begin{aligned} \text{Expressions } e ::= & x \mid c \mid \lambda x : \tau. e \mid e_1 e_2 \mid \text{let } x = e_1 \text{ in } e_2 \\ & \mid (\lambda x : \tau_1. e : \tau_2) \text{ by } m \\ & \mid \{l_i = e_i\}_{i \in I} \mid e.l \\ & \mid [e_1, \dots, e_n] \mid \{e_1 : e'_1, \dots, e_n : e'_n\} \end{aligned}$$

Notez que byLLM supporte plus de types que ceux définis ci-dessus (par exemple, les énums, les dictionnaires, les tuples, etc.). Cependant, pour simplifier la formalisation, nous nous contenterons de ces types.

Les types unions sont aussi supportés, cependant la formalisation est plus complexe nécessitant des sum types que nous ne traiterons pas ici.

Les contextes de typages sont définis comme suit :

- Γ est un contexte de typage pour les variables, c'est-à-dire un ensemble de paires (x, τ) où x est une variable et τ son type.
- M est un contexte de modèles, c'est-à-dire un ensemble de modèles m disponibles.

Notez aussi que nous ne typons pas les modèles eux-mêmes. En effet, tous les LLMs prennent et retournent des chaînes de caractères. Notre système vérifie plutôt la cohérence entre les annotations de types et la génération de schémas JSON.

2.2. Relation types-schémas JSON

Posons $\llbracket \tau \rrbracket$ la relation qui associe à un type τ le schéma JSON correspondant. Cette relation est définie de la manière suivante :

$$\begin{aligned} \llbracket \text{int} \rrbracket &= \{ "type" : "integer" \} \\ \llbracket \text{str} \rrbracket &= \{ "type" : "string" \} \\ \llbracket \text{bool} \rrbracket &= \{ "type" : "boolean" \} \\ \llbracket \text{float} \rrbracket &= \{ "type" : "number" \} \\ \llbracket \text{None} \rrbracket &= \{ "type" : "null" \} \\ \llbracket \text{List}\langle\tau\rangle \rrbracket &= \{ "type" : "array", "items" : \llbracket \tau \rrbracket \} \\ \llbracket \text{Record}\{\ell_i : \tau_i\}_{i \in I} \rrbracket &= \{ "type" : "object", "properties" : \{ \ell_i : \llbracket \tau_i \rrbracket \}_{i \in I} \} \end{aligned}$$

Les transformation sont directement implémentées dans la fonction `_type_to_schema` du fichier `jac-byllm/byllm/schema.py` du dépôt de JacLang.

On remarquera que la fonction rajoute aussi des informations supplémentaires aux schémas tel que les éventuels titre et description. Ces informations sont extraites du MT-IR en string donc nous y ferons abstraction car ne risque pas de poser des problèmes de cohérence.

2.3. Règles de typage

Les règles de typage de MTP sont définies comme suit :

Règles de base

$$\frac{x : \tau \in \Gamma}{\Gamma; M \vdash x : \tau} (\text{T-Var}) \quad \frac{}{\Gamma; M \vdash c : \text{typeOf}(c)} (\text{T-B})$$

$$\frac{\Gamma, x : \tau_1; M \vdash e : \tau_2}{\Gamma; M \vdash \lambda x : \tau_1. e : \tau_1 \rightarrow \tau_2} (\text{T-Abs})$$

$$\frac{\Gamma; M \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma; M \vdash e_2 : \tau_1}{\Gamma; M \vdash e_1 e_2 : \tau_2} (\text{T-App})$$

Pour les types de bases, nous avons une règle par type comme la règle T-B.

Règles pour les types structurés

Pour les records :

$$\frac{\Gamma; M \vdash e_i : \tau_i \quad \forall i \in I}{\Gamma; M \vdash \{l_i = e_i\}_{i \in I} : \text{Record}\{l_i : \tau_i\}_{i \in I}} (\text{T-Record})$$

$$\frac{\Gamma; M \vdash e : \text{Record}\{l_i : \tau_i\}_{i \in I} \quad j \in I}{\Gamma; M \vdash e.l_j : \tau_j} (\text{T-Proj})$$

Pour les listes homogènes :

$$\frac{\Gamma; M \vdash e_i : \tau \quad \forall i \in \{1, \dots, n\}}{\Gamma; M \vdash [e_1, \dots, e_n] : \text{List}\langle \tau \rangle} (\text{T-List})$$

Règle spécifique à MTP

$$\frac{\Gamma; M \vdash \lambda x : \tau_1. e : \tau_1 \rightarrow \tau_2 \quad m \in M \quad \llbracket \tau_1 \rrbracket \text{ existe} \quad \llbracket \tau_2 \rrbracket \text{ existe}}{\Gamma; M \vdash (\lambda x : \tau_1. e : \tau_2) \text{ by } m : \tau_1 \rightarrow \tau_2} (\text{T-ByLLM})$$

Cette règle signifie que `by m` décore une fonction pour que son implémentation soit fournie par le modèle `m`. Le résultat est une fonction de même type que la fonction originale.

Les conditions décrites sont les suivantes :

1. L'expression doit être une fonction de type $\tau_1 \rightarrow \tau_2$.
2. Le modèle `m` doit être connu.
3. le type τ_1 doit pouvoir être converti en un schéma JSON (pour générer le prompt).
4. le type τ_2 doit pouvoir être converti en un schéma JSON (pour valider la sortie).

En pratique, byLLM supporte des fonctions avec un nombre arbitraire de paramètres. Dans notre système par curryfication, ceci revient à des fonctions de type $\tau_1 \rightarrow (\tau_2 \rightarrow (\dots \rightarrow \tau_n))$.

2.4. Sémantique opérationnelle

β réduction

$$\frac{}{(\lambda x : \tau.M) N \rightarrow M[N/x]} (\beta\text{-reduction})$$

μ transformation

$$\begin{array}{c} \frac{M \rightarrow M'}{M N \rightarrow M' N} (\mu_1\text{-transformation}) \quad \frac{N \rightarrow N'}{M N \rightarrow M N'} (\mu_2\text{-transformation}) \\ \\ \frac{e_k \rightarrow e'_k}{\{l_1 = e_1, \dots, l_k = e_k, \dots, l_n = e_n\} \rightarrow \{l_1 = e_1, \dots, l_k = e'_k, \dots, l_n = e_n\}} (\mu_{\text{Record}}\text{-transformation}) \\ \\ \frac{e_i \rightarrow e'_i}{[e_1, \dots, e_i, \dots, e_n] \rightarrow [e_1, \dots, e'_i, \dots, e_n]} (\mu_{\text{List}}\text{-transformation}) \end{array}$$

ξ congruence

$$\frac{M \rightarrow M'}{\lambda x.M \rightarrow \lambda x.M'} (\xi\text{-congruence}) \quad \frac{e \rightarrow e'}{e.l \rightarrow e'.l} (\xi_{\text{Record}}\text{-congruence})$$

Réduction des invocation LLM

De part la nature non-déterministe des LLMs, il est impossible de définir une sémantique opérationnelle déterministe pour MTP. Ici nous nous contenterons de dire que 2 réductions sont possibles :

$$\begin{array}{c} v : \tau_1 \quad (\lambda x : \tau_1.e : \tau_2) \text{ by } m : \tau_1 \rightarrow \tau_2 \\ \text{json_str} = \text{invoke_llm}(m, \lambda x : \tau_1.e, v, \llbracket \tau_2 \rrbracket) \\ \frac{v' = \text{validate}(\text{json_str}, \tau_2)}{((\lambda x : \tau_1.e : \tau_2) \text{ by } m) v \rightarrow v'} (\beta\text{-ByLLM-Success}) \\ \\ \text{json_str} = \text{invoke_llm}(m, \lambda x : \tau_1.e, v, \llbracket \tau_2 \rrbracket) \\ \text{validate}(\text{json_str}, \tau_2) = \text{error} \\ \frac{}{((\lambda x : \tau_1.e : \tau_2) \text{ by } m) v \rightarrow \text{ValidationError}} (\beta\text{-ByLLM-Fail}) \end{array}$$

où :

- $\text{invoke_llm}(m, \lambda x : \tau_1.e, v, S)$ appelle le LLM m avec :
 - Un prompt généré à partir de la fonction et de la valeur d'entrée v
 - Le schéma JSON $S = \llbracket \tau_2 \rrbracket$ pour contraindre la sortie
 - Retourne une chaîne JSON
- $\text{validate}(\text{json_str}, \tau_2)$ parse et valide le JSON selon le type τ_2 (via Pydantic)

3. Propriétés

3.1. Type Safety

Montrons que le système de type de MTP garantit la sûreté de type. Plus précisément, montrons que pour tout programme P typable, l'exécution de P ne lève jamais d'erreur de type. Pour ce faire, nous devons montrer la progression et la préservation du typage.

Pour cette démonstration nous supposons que les LLMs sont stables et que la validation réussit toujours. Donc que la règle $\beta\text{-ByLLM-Fail}$ ne peut pas se produire.

3.1.1. Progression

Nous devons montrer que si $\vdash t : T$ et t est fermé alors soit t est une valeur, soit il existe t' tel que $t \rightarrow t'$.

Par induction sur la forme de t :

1. **Cas t est une constante** : t est une valeur de type B avec $B \in \{Int, Str, Bool, Float, None\}$.
2. **Cas $t = \lambda x.e$** : t est une valeur.
3. **Cas $t = ((\lambda x : \tau_1.e : \tau_2) \text{ by } m)$** : t est une valeur de type $\tau_1 \rightarrow \tau_2$.
4. **Cas $t = [e_1, \dots, e_n]$** : Par hypothèse d'induction :
 - Soit $\exists k$ tel que $e_k \rightarrow e'_k$, auquel cas nous pouvons réduire avec μ_{List} .
 - Soit $\forall k, e_k$ est une valeur, auquel cas t est une valeur.
5. **Cas $t = l_1 = e_1, \dots, l_n = e_n$** : Par hypothèse d'induction :
 - Soit $\exists k$ tel que $e_k \rightarrow e'_k$, auquel cas nous pouvons réduire avec μ_{Record} .
 - Soit $\forall k, e_k$ est une valeur, auquel cas t est une valeur.
6. **Cas $t = e_1.l$** : on a donc $\vdash e_1 : \text{Record}\{l_i : \tau_i\}_{i \in I}$. Par hypothèse d'induction, soit :
 - $e_1 \rightarrow e'_1$, auquel cas nous pouvons réduire avec ξ_{Record} .
 - e_1 est une valeur, auquel cas t est une valeur.
7. **Cas $t = e_1 e_2$** : on a donc $\vdash e_1 : \tau_1 \rightarrow \tau_2$ et $\vdash e_2 : \tau_1$. Donc soit :
 - e_1 est d'expression $\lambda x.e$, donc par hypothèse d'induction :
 - Soit e_1 et e_2 sont des valeurs, auquel cas nous pouvons réduire avec β -réduction.
 - Soit $e_2 \rightarrow e'_2$, auquel cas nous pouvons réduire avec μ_2 .
 - Soit $e_1 \rightarrow e'_1$, auquel cas nous pouvons réduire avec μ_1 .
 - Soit les deux derniers cas se cumulent, auquel cas nous pouvons réduire avec μ_1 ou μ_2 .
 - e_1 est d'expression $((\lambda x : \tau_1.e : \tau_2) \text{ by } m)$, donc par hypothèse d'induction :
 - Soit e_1 et e_2 sont des valeurs, auquel cas nous pouvons réduire soit avec **E-ByLLM-Success**.
 - Les autres cas sont les mêmes que pour le cas précédent.

3.1.2. Préservation du typage

Nous devons montrer que si $\Gamma; M \vdash t : \tau$ et $t \rightarrow t'$ alors $\Gamma; M \vdash t' : \tau$.

On pose le lemme suivant :

Lemme de substitution : Si $\Gamma, x : \tau_1; M \vdash e : \tau_2$ et $\Gamma; M \vdash v : \tau_1$, alors $\Gamma; M \vdash e[v/x] : \tau_2$.

Par induction sur la dérivation de $t \rightarrow t'$:

1. **Cas β -réduction** : $(\lambda x : \tau_1.e) v \rightarrow e[v/x]$ avec $\Gamma; M \vdash \lambda x : \tau_1.e : \tau_1 \rightarrow \tau_2$ et $\Gamma; M \vdash v : \tau_1$.
 - Par inversion de T-Abs, on a $\Gamma, x : \tau_1; M \vdash e : \tau_2$.
 - Par le lemme de substitution, $\Gamma; M \vdash e[v/x] : \tau_2$.
2. **Cas μ_1 -transformation** : $e_1 e_2 \rightarrow e'_1 e_2$ avec $e_1 \rightarrow e'_1$.
 - On a $\Gamma; M \vdash e_1 : \tau_1 \rightarrow \tau_2$ et $\Gamma; M \vdash e_2 : \tau_1$.
 - Par hypothèse d'induction, $\Gamma; M \vdash e'_1 : \tau_1 \rightarrow \tau_2$.
 - Par T-App, $\Gamma; M \vdash e'_1 e_2 : \tau_2$.
3. **Cas μ_2 -transformation** : similaire à μ_1 .
4. **Cas μ_{List} -transformation** : $[e_1, \dots, e_i, \dots, e_n] \rightarrow [e_1, \dots, e'_i, \dots, e_n]$ avec $e_i \rightarrow e'_i$.
 - On a $\Gamma; M \vdash e_j : \tau$ pour tout $j \in \{1, \dots, n\}$.
 - Par hypothèse d'induction, $\Gamma; M \vdash e'_i : \tau$.
 - Par T-List, $\Gamma; M \vdash [e_1, \dots, e'_i, \dots, e_n] : \text{List}(\tau)$.
5. **Cas μ_{Record} -transformation** : similaire à μ_{List} .
6. **Cas ξ_{Record} -congruence** : $e.l \rightarrow e'.l$ avec $e \rightarrow e'$.

On a $\Gamma; M \vdash e : \text{Record}\{l_i : \tau_i\}_{i \in I}$ avec $l \in \{l_i\}_{i \in I}$.

Par hypothèse d'induction, $\Gamma; M \vdash e' : \text{Record}\{l_i : \tau_i\}_{i \in I}$.

Par T-Proj, $\Gamma; M \vdash e'.l : \tau_l$.

7. **Cas β -ByLLM-Success** : $((\lambda x : \tau_1.e : \tau_2) \text{ by } m) v \rightarrow v'$ où $v' = \text{validate}(\text{invoke_llm}(m, \lambda x : \tau_1.e, v, [\![\tau_2]\!]), \tau_2)$.

On a $\Gamma; M \vdash (\lambda x : \tau_1.e : \tau_2) \text{ by } m : \tau_1 \rightarrow \tau_2$ et $\Gamma; M \vdash v : \tau_1$.

Par T-App, le type de retour est τ_2 .

Par définition de validate, si la validation réussit, alors v' est bien typé selon τ_2 (via Pydantic).

Donc $\Gamma; M \vdash v' : \tau_2$.

Preuve : Par induction sur la dérivation de $\Gamma, x : \tau_1; M \vdash e : \tau_2$.

3.2. Soundness de la traduction type-schéma

Au-delà de la sûreté de type classique, le système MTP doit garantir la **correspondance sémantique** entre trois niveaux d'abstraction : les types déclarés statiquement, les schémas JSON générés pour contraindre les LLMs, et la validation des sorties à l'exécution. Cette section établit formellement que la traduction $[\![\cdot]\!]$ préserve la sémantique des types et est utilisée de manière uniforme à travers toutes les phases du système.

Clarifions d'abord la nature de la relation $[\![\cdot]\!]$: il ne s'agit pas d'une règle de réduction opérationnelle (comme \rightarrow), mais d'une **fonction de traduction statique** qui associe à chaque type un schéma JSON. Cette fonction est utilisée à deux moments distincts : lors du typage (règle T-ByLLM) pour vérifier que les types peuvent être traduits, et lors de l'exécution (règles β -ByLLM-Success et β -ByLLM-Fail) pour générer les schémas passés au LLM. La soundness du système repose sur le fait que la **même fonction $[\![\cdot]\!]$ est utilisée partout** et qu'elle capture fidèlement la sémantique des types.

3.2.1. Propriétés de la fonction de traduction

Nous établissons d'abord que $[\![\cdot]\!]$ est une fonction totale et déterministe, garantissant qu'elle peut être utilisée de manière fiable dans toutes les phases du système.

Lemme 1 (Totalité de la traduction) : Pour tout type τ bien formé utilisable dans une expression $(\lambda x : \tau_1.e : \tau_2) \text{ by } m$ bien typée selon T-ByLLM, il existe un schéma JSON $[\![\tau]\!]$.

Preuve : Par construction de la règle T-ByLLM, qui exige explicitement que $[\![\tau_1]\!]$ et $[\![\tau_2]\!]$ existent. La fonction $[\![\cdot]\!]$ est définie inductivement sur la structure des types (section 2.2), couvrant tous les types de base B et les constructeurs de types (List, Record). Par induction structurelle sur τ :

- **Cas de base** : Pour tout type de base $B \in \{\text{int}, \text{str}, \text{bool}, \text{float}, \text{None}\}$, $[\![B]\!]$ est défini explicitement.
- **Cas inductif** :
 - Pour List $\langle \tau' \rangle$, si $[\![\tau']\!]$ existe par hypothèse d'induction, alors $[\![\text{List}\langle \tau' \rangle]\!]$ existe.
 - Pour Record $\{l_i : \tau_i\}_{i \in I}$, si $[\![\tau_i]\!]$ existe pour tout $i \in I$ par hypothèse d'induction, alors $[\![\text{Record}\{l_i : \tau_i\}_{i \in I}]\!]$ existe.

Lemme 2 (Déterminisme de la traduction) : Pour tout type τ , la fonction $[\![\cdot]\!]$ produit un unique schéma JSON.

Preuve : Par induction sur la structure de τ . Chaque cas de la définition de $[\![\cdot]\!]$ (section 2.2) est une fonction déterministe qui associe à chaque type exactement un schéma JSON. Il n'y a pas d'ambiguïté dans la définition.

Ces deux lemmes garantissent qu'il n'existe pas de désynchronisation entre le type annoté par le développeur et le schéma JSON transmis au LLM. Pour un type donné, le schéma généré est toujours identique, quelle que soit la phase (typage ou exécution).

3.2.2. Soundness de la validation

La deuxième garantie établit que la validation des sorties LLM préserve la conformité aux types, c'est-à-dire que les schémas JSON capturent fidèlement la sémantique des types MTP.

Lemme 3 (Soundness de la validation) : Si $\text{validate}(\text{json_str}, \tau) = v$ alors $\vdash v : \tau$.

Preuve : La fonction validate est implémentée via Pydantic, qui effectue une vérification structurelle complète basée sur le schéma $\llbracket \tau \rrbracket$. Nous montrons par induction sur la structure de τ que toute valeur acceptée par le schéma $\llbracket \tau \rrbracket$ est bien typée selon τ :

- **Cas de base** : Pour les types de base B , la validation vérifie que la valeur JSON correspond au type primitif attendu (entier pour int, chaîne pour str, etc.). Si la validation réussit, alors v est bien une constante de type B par construction.
- **Cas inductif** :
 - Pour $\text{List}\langle \tau' \rangle$, le schéma $\llbracket \text{List}\langle \tau' \rangle \rrbracket$ exige un tableau JSON dont chaque élément satisfait $\llbracket \tau' \rrbracket$. La validation applique récursivement la validation à chaque élément selon τ' . Par hypothèse d'induction, chaque élément validé est de type τ' , donc v est de type $\text{List}\langle \tau' \rangle$ par T-List.
 - Pour $\text{Record}\{l_i : \tau_i\}_{i \in I}$, le schéma $\llbracket \text{Record}\{l_i : \tau_i\}_{i \in I} \rrbracket$ exige un objet JSON possédant exactement les champs l_i avec des valeurs satisfaisant $\llbracket \tau_i \rrbracket$. Par hypothèse d'induction, chaque champ validé est de type τ_i , donc v est de type $\text{Record}\{l_i : \tau_i\}_{i \in I}$ par T-Record.

Ce lemme établit qu'il n'existe pas de faux positifs dans la validation : toute sortie LLM acceptée par le système est garantie conforme au type déclaré. Autrement dit, les schémas JSON générés par $\llbracket \cdot \rrbracket$ capturent exactement la sémantique des types MTP.

3.2.3. Uniformité de la traduction

La troisième garantie établit que les vérifications statiques et dynamiques utilisent la même fonction de traduction, assurant ainsi qu'il n'y a pas de divergence entre les contraintes vérifiées au typage et celles appliquées à l'exécution.

Lemme 4 (Uniformité statique-dynamique) : Pour toute expression $(\lambda x : \tau_1.e : \tau_2)$ by m bien typée selon T-ByLLM, les schémas utilisés dans les règles de réduction β -ByLLM-Success et β -ByLLM-Fail sont exactement $\llbracket \tau_1 \rrbracket$ et $\llbracket \tau_2 \rrbracket$.

Preuve : Par inspection des règles de typage et de réduction :

- La règle T-ByLLM vérifie l'existence de $\llbracket \tau_1 \rrbracket$ et $\llbracket \tau_2 \rrbracket$ lors du typage statique.
- Les règles β -ByLLM-Success et β -ByLLM-Fail utilisent explicitement $\llbracket \tau_2 \rrbracket$ dans l'invocation $\text{invoke_llm}(m, \lambda x : \tau_1.e, v, \llbracket \tau_2 \rrbracket)$ et dans la validation $\text{validate}(\text{json_str}, \tau_2)$.
- Par déterminisme de $\llbracket \cdot \rrbracket$ (Lemme 2), pour un type τ_2 donné, le schéma $\llbracket \tau_2 \rrbracket$ calculé lors du typage est identique à celui utilisé lors de l'exécution.
- Aucune autre fonction de traduction n'est utilisée dans le système.

Ce lemme garantit qu'il n'existe pas de divergence entre ce qui est vérifié statiquement et ce qui est appliqué dynamiquement. Les contraintes de types sont uniformes à travers toutes les phases d'exécution.

3.2.4. Théorème principal de soundness

Nous établissons maintenant le résultat principal qui combine les garanties précédentes et montre que le système MTP est sound de bout en bout.

Théorème (Soundness de bout en bout) : Pour toute expression e bien typée avec $\Gamma; M \vdash e : \tau$ contenant des sous-expressions by m , si $e \rightarrow^* v$ où v est une valeur et aucune réduction n'a produit de ValidationError, alors $\Gamma; M \vdash v : \tau$.

Preuve : Par induction sur le nombre de pas de réduction $e \rightarrow^* v$:

- **Cas de base** : Si $e = v$, alors le résultat est immédiat par hypothèse.
- **Cas inductif** : Si $e \rightarrow e' \rightarrow^* v$, alors :

- Par préservation du typage (Théorème 3.1.2), $\Gamma; M \vdash e' : \tau$.
- Si la réduction $e \rightarrow e'$ utilise β -ByLLM-Success, alors :
 - Le schéma utilisé est $\llbracket \tau_2 \rrbracket$ par construction de la règle.
 - Par le Lemme 4 (uniformité), ce schéma est identique à celui vérifié lors du typage.
 - La validation $\text{validate}(\text{json_str}, \tau_2)$ produit une valeur v' .
 - Par le Lemme 3 (soundness de la validation), $\Gamma; M \vdash v' : \tau_2$.
- Si la réduction $e \rightarrow e'$ utilise une autre règle, la préservation du typage s'applique directement.
- Par hypothèse d'induction sur $e' \rightarrow^* v$, on obtient $\Gamma; M \vdash v : \tau$.

Ce théorème établit que le système MTP garantit la sûreté de type de bout en bout, même en présence d'appels LLM non-déterministes, tant que ces appels réussissent leur validation. Les erreurs de type ne peuvent survenir que lors d'échecs de validation explicites (ValidationError), qui sont détectés et signalés par le système avant toute utilisation de la valeur invalide. La traduction $\llbracket \cdot \rrbracket$ assure que les contraintes de types sont fidèlement représentées dans les schémas JSON et uniformément appliquées à travers toutes les phases du système.

4. Conclusion de la formalisation

4.1. Résumé des contributions

Dans cette seconde partie, nous avons proposé une formalisation du paradigme Meaning-Typed Programming (MTP) sous la forme d'un système de types avec sémantique opérationnelle. Cette formalisation permet de raisonner de manière abstraite sur les propriétés du système, indépendamment de son implémentation dans le langage Jac.

Le système formel comprend une syntaxe pour les types et expressions intégrant l'opérateur `by`, une fonction de traduction type-schéma $\llbracket \cdot \rrbracket$ traduisant les types en schémas JSON, des règles de typage étendant un système classique avec la règle T-ByLLM, et une sémantique opérationnelle modélisant l'exécution via les règles de réduction classiques (β , μ , ξ) et les règles spécifiques aux invocations LLM (β -ByLLM-Success et β -ByLLM-Fail).

Nous avons établi plusieurs propriétés fondamentales du système MTP. La sûreté de type est garantie par les propriétés de progression et de préservation (section 3.1), démontrant que toute expression bien typée et fermée est soit une valeur, soit peut être réduite, et que la réduction préserve les types. La soundness du système est assurée par quatre lemmes (section 3.2) : la totalité et le déterminisme de la fonction de traduction type-schéma, la soundness de la validation (les schémas capturent fidèlement la sémantique des types), l'uniformité de la traduction entre phases statiques et dynamiques, et le théorème principal de soundness de bout en bout établissant que les valeurs produites respectent toujours leurs types déclarés.

4.2. Limitations de la formalisation

Notre formalisation présente plusieurs limitations qu'il convient de reconnaître. Pour simplifier la présentation, nous avons omis plusieurs types supportés par l'implémentation réelle de byLLM, notamment les types union, les énumérations, les dictionnaires avec clés dynamiques et les tuples. De même, bien que notre syntaxe inclue le polymorphisme paramétrique $(\forall \alpha. \tau)$, nous n'avons pas défini les règles de typage correspondantes, l'extension complète nécessitant de traiter la génération de schémas pour les types polymorphes.

La modélisation des LLMs reste également abstraite. Nous avons représenté l'invocation LLM comme une fonction `invoke_llm` sans spécifier la structure exacte des prompts générés, les mécanismes de retry ou les stratégies de gestion d'erreurs. Notre sémantique opérationnelle capture le non-déterminisme via deux règles de réduction (Success/Fail), mais ne modélise ni les distributions de probabilité sur les sorties, ni les garanties probabilistes de correction, ni l'impact des hyperparamètres.

Enfin, plusieurs aspects centraux du fonctionnement de MTP ne sont pas formalisés : le processus d'extraction des informations sémantiques (MT-IR), la transformation de la sémantique en prompts textuels, et les aspects de coûts et performances (tokens, latence, caching, batching). Ces éléments relèvent davantage de l'analyse de programmes et de l'ingénierie que de la théorie des types.

4.3. Extensions possibles

Plusieurs directions permettraient d'enrichir et de renforcer cette formalisation. L'ajout de types union ($\tau_1 \cup \tau_2$) et intersection ($\tau_1 \cap \tau_2$) nécessiterait des règles de sous-typage et une extension de la fonction $\llbracket \cdot \rrbracket$ pour traduire ces types en schémas JSON (via `anyOf`, `allOf`), ainsi qu'une preuve de soundness pour ces nouveaux constructeurs. Des types dépendants permettraient d'exprimer des contraintes sur les valeurs, tandis qu'un système de types avec effets ou monades modéliserait explicitement le non-déterminisme, l'I/O et les erreurs des appels LLM.

Une sémantique probabiliste, où chaque réduction $t \rightarrow t'$ serait associée à une probabilité et les propriétés de sûreté exprimées en termes probabilistes, permettrait de raisonner formellement sur la fiabilité des applications MTP. La mécanisation des preuves dans un assistant comme Coq, Isabelle ou Lean vérifierait rigoureusement leur correction et permettrait l'extraction d'une implémentation certifiée de MT-Runtime. Enfin, des analyses statiques estimant le nombre de tokens consommés ou détectant les patterns inefficaces aideraient les développeurs à optimiser leurs applications.

Une extension particulièrement intéressante serait d'établir la **completeness** de la traduction type-schéma : montrer que si une valeur v est de type τ , alors elle satisfait nécessairement le schéma $\llbracket \tau \rrbracket$. Combinée avec la soundness (Lemme 3), cette propriété établirait une équivalence sémantique complète entre types et schémas, renforçant ainsi les garanties du système.

Partie III : jacEMMA - Creation d'un Projet original

Prediction de trajectoire de véhicule multimodale avec byllm()

1. Introduction

L'objectif de notre démarche est d'adopter le point de vue d'un développeur souhaitant intégrer la bibliothèque à son projet. Nous commençons par implémenter quelques applications simples pour prendre en main la syntaxe de `jac` et de `byllm` (un traducteur, un générateur de niveaux) et l'opérateur `by`.

Nous testons ensuite les fonctionnalités multimodales de `byllm()` à travers un projet de prédiction de trajectoire future d'un véhicule. Finalement, nous comparerons ce programme au même réalisé en Python, en utilisant des techniques plus traditionnelles, selon différentes métriques.

Contributions :

- Exemples simples d'utilisation de `jac/byllm` et de l'opérateur `by`.
- Présentation d'un traducteur et d'un générateur de profils d'étudiant implémentée avec `jac/byllm` et opérateur `by`
- `TrajectoryPrediction`
 - Prototype multimodal utilisant `byllm()`. Prédit les trajectoires futures à partir des informations passées de l'ego d'un véhicule et d'une photo frontale.
- `Benchmark`
 - Comparaison qualitative (LoC, Temps d'inference, évaluation de l'output)
 - Reflexions personnelles.

2. Prise En Main

Par quelques cas d'utilisation simples, on explore les fonctionnalités de `jac` et `byllm()`.

`translator.jac` (par Function Definitions)

On souhaite traduire une phrase vers une langue donnée.

```
def translate_to(language: str, phrase: str) -> str by llm()
```

Appel de la fonction :

```
with entry {
    lang = "French";
    phrase = "Hello, how are you?";
    output = translate_to(language=lang, phrase=phrase);
    print(output);
}
```

En suivant l'algorithme MT-IR et la synthèse de prompt de MTP, le prompt généré par la sémantique du code (lors de l'appel de la fonction seulement) serait comme dans la Figure 1.

stdin :

```
Bonjour, comment allez-vous ?
```

`studentGenerator.jac` (par Function Definitions et Member Method Definition)

Ce programme, plus complexe, génère un étudiant et détermine son âge à partir de sa date de naissance. On a 3 objets : `Student`, `University` et `Class`, avec un enum représentant le genre.

```
obj University {
    has name: str;
    has location: str;
    has ranking: int;
}
```

```

[System Prompt]
This is an operation you must perform and return the output values.

[Inputs_Information]
(language) (str) = "French"
(phrase) (str) = "Hello, how are you?"

[Output_Information]
(str)

[Type_Explanations]
(str)

[Action]
translate_to
Generate and return the output result(s) only, adhering to the provided Type in the
following format
[Output]
<results>

```

FIGURE 1 – Prompt généré pour l’appel de ‘translate_to’

```

obj Class{
    has class_name: str;
    has professor: str;
    has credits: int;
}

enum Gender {
    MALE,
    FEMALE,
    OTHER
}

obj Student {
    has name: str;
    has date_of_birth: str;
    has gender : Gender;
    has grade: str;
    has classes: list[Class];
    has university: University;
    def introduce() -> str by llm();
}

```

On remarque la méthode `Student.introduce()` qui délègue la génération du texte au modèle via `llm()`, produisant une courte présentation de l’étudiant. Le mot-clé `by` sur une méthode extrait également le contexte sémantique de son objet. On ajoute deux fonctions : `generate_student(gender: Gender)` qui génère un `Student` selon son genre, et `calculate_age(student: Student)` qui calcule l’âge de l’étudiant à partir de sa date de naissance.

```
def generate_student(gender: gender) -> Student by llm(temperature=0.4);
```

Pour que la fonction de calcul de l’âge fonctionne correctement, on peut inclure des informations supplémentaires dans `llm()`, ici la date du jour.

```
def calculate_age(Student: Student) -> int by llm(incl_info={
    "today": datetime.now()});

```

On génère un étudiant homme, on calcule son âge, et on l’introduit.

```

let student_m = generate_student(gender=gender.MALE);
let age_m = calculate_age(Student=student_m);
print("##### student introduction #####");
print(student_m.introduce());

```

```

print("##### student age #####");
print(f"student age: {age_m}, studen date of birth: {student_m.date_of_birth}");
print("##### student info #####");
print(f"student info: {student_m}");

```

Exemple d'output :

```

##### student introduction #####
Student 'John Doe' born on '1990-01-01',
male,
in 12th grade at University of Technology (ranked 10) in New York.
#####
student age #####
student age: 35, student date of birth: 1990-01-01
#####
student info #####
student info: Student(name='John Doe',
                      date_of_birth='1990-01-01',
                      sex=<sex.MALE: 1>,
                      grade='12th',
                      classes=[],
                      university=University(name='University of Technology',
                                           location='New York', ranking=10))

```

En suivant l'algorithme MT-IR et la synthèse de prompt de MTP, le prompt généré par la sémantique de l'appel de `generate_student` ressemblerait à la Figure 2.

```

[System Prompt]
This is an operation you must perform and return the output values.

[Inputs_Information]

(gender) (Gender) = MALE

[Output_Information]
(Student)

[Type_Explanations]
(Student)(obj)eg:
- Student(name:str,gender:Gender...,classes:list[Class],university:University)
(Gender) (enum)eg:
- Gender(MALE,FEMALE,OTHER)
(Class)(obj)eg:
- Class(class_name:str,professor:str,credits:int)
(University)(obj)eg:
- University(name:str,location:str,ranking:int)

[Action]
generate_student
Generate and return the output result(s) only, adhering to the provided Type in the
following format
[Output]
<result>

```

FIGURE 2 – Prompt généré pour l'appel de ‘generate_student’

3. Prototype : JacEMMA - Prédiction de trajectoire de véhicule

L'objectif de notre prototype est de prédire la trajectoire future d'un véhicule en utilisant sa vitesse et sa courbure passées ainsi qu'une image capturée par une caméra frontale.

L'idée est venue de la méthode LightEMMA (End to End Multi-Modal Attention for Vehicle Trajectory Prediction) qui utilise des modèles multimodaux pour prédire des trajectoires d'un véhicule à partir de sa caméra frontale et d'informations temporelles sur l'ego du véhicule. C'est une tâche complexe qui nécessite un prompt engineering poussé pour obtenir un résultat correct. Ce projet permettra de pousser les capacités sémantiques de `byllm()` pour une tâche relativement complexe.

3.1. Méthode

Deux objectifs principaux dans notre projet :

- répliquer le comportement de LightEMMA sur une frame d'une situation de conduite
- comparer les résultats d'une implémentation en Jac avec `byllm()` par rapport à une implémentation classique en Python.

Là où le modèle utilise la librairie Transformers pour charger et inférer un modèle multimodal, nous allons utiliser `byllm()` interrogeant un modèle local via Ollama. L'implémentation classique utilisera la librairie Ollama pour faire des appels au modèle local.

LightEMMA requête deux fois le VLM pour intégrer une description de la scène et des intentions de conduite dans le prompt final. Nous utiliserons un seul prompt unique pour la méthode traditionnelle.

Bien que la philosophie de `byllm` soit d'extraire la sémantique du code pour former un prompt automatiquement, le langage Jac propose le type `Semstrings` qui permet d'inclure des informations supplémentaires au sujet de variables, fonctions, attributs ou objets. Nous comparerons les résultats avec et sans ces Semstrings pour évaluer leur impact ; nous testerons également une version contenant un prompt manuel passé à la fonction de prédiction comme attribut d'objet.

Formellement, on définit P comme la fonction de prédiction de trajectoire future :

$$P(I, \mathbf{v}_{1:t}, \kappa_{1:t}, s) = [(v_{t+1}, \kappa_{t+1}), \dots, (v_{t+6}, \kappa_{t+6})],$$

- I : image frontale
- $\mathbf{v}_{1:t}$: vitesses passées
- $\kappa_{1:t}$: courbures passées
- s : contexte supplémentaire optionnel (Semstrings ou prompt manuel)

et la sortie est une liste de 6 tuples (driving actions) représentant la vitesse et la courbure prévues pour les 3 prochaines secondes (0,5 s d'intervalle) :

$$(v_{t+k}, \kappa_{t+k}) \in \mathbb{R}^2, \quad k = 1, \dots, 6.$$

- Pour faciliter la notation, on définit :
 - P_{ss} : version de P avec Semstrings (contexte sémantique ajouté)
 - P_{pm} : version de P avec prompt manuel (prompt passé comme attribut)
 - P_{def} : version de P sans contexte supplémentaire (défaut, philosophie du papier original)
 - P_{py} : implémentation Python classique sans `byllm()`

Bien que P_{pm} et P_{py} utilisent le même prompt, il est intéressant de comparer leur robustesse lors de l'évaluation de l'output.

Métriques

Pour évaluer les performances de chaque version, nous convertissons l'output en une liste de waypoints et utilisons les métriques suivantes :

Précision de la trajectoire

- **ADE** (Average Displacement Error) : erreur moyenne entre les positions prédites et réelles (voir Figure 3).
- **FDE** (Final Displacement Error) : erreur sur la position finale.
- **L2 error** : distance moyenne point à point entre prédictions et valeurs réelles.

Efficacité

- Temps d'inférence moyen par appel de fonction.
- Nombre de lignes de code (LoC) pour chaque version (P et structures de données seulement ; ce n'est pas la même métrique utilisée dans l'article).

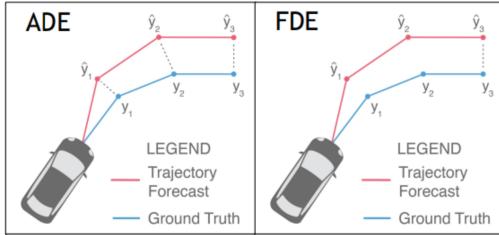


FIGURE 3 – Comparaison ADE/FDE. Source : Rethinking Trajectory Forecasting Evaluation - Boris Ivanovic, Marco Pavone

Robustesse

- Pourcentage d'erreurs de parsing de la sortie.

3.2 Choix du modèle

Modèle multimodal local qwen2.5VL_8b via Ollama.

Assez petit pour inférer rapidement sur une machine locale, c'est également un modèle déjà benchmarké pour des tâches de prédiction de trajectoire sur le même dataset.

Par contrainte temporelle, nous n'avons pas pu tester d'autres modèles, mais il serait intéressant de comparer les performances de qwen2.5VL_8b avec des modèles plus puissants, je pense au modèle Cosmos-Reason1-7B qui a montré des performances supérieures sur ce genre de tâche avec un prompt unique (Figure).

TABLE III: Subsets used

Model	Prompt system	Temp	Top p	Rep p	FDE	ADE
Qwen2.5 VL-7B-Instruct	Legacy	0.8	0.8	1.0	2.307	1.124
	Structured	0.6	0.95	1.05	2.411	1.185
	One	0.4	0.8	1.0	2.816	1.446
Cosmos Reason1-7B	Legacy	0.4	0.8	1.05	2.939	1.445
	Structured	0.4	0.8	1.05	2.945	1.517
	One	0.4	0.8	1.0	2.500	1.231

FIGURE 4

3.2. Choix d'Évaluation

Données d'évaluation

Nous utilisons une frame du dataset nuScenes avec la liste de ses vitesses et courbures passées pendant 3 s à 0,5 s d'intervalle (6 valeurs) ainsi qu'une photo capturée par la caméra avant du véhicule (voir Figure 6).

50 essais ont été réalisés pour P , P_{pm} , P_{def} et P_{py} .

TABLE III: Subsets used						
Model	Prompt system	Temp	Top n	Rep n	FDE	ADE
Qwen2.5 VL-7B-Instruct	Legacy	0.8	0.8	1.0	2.307	1.124
	Structured	0.6	0.95	1.05	2.411	1.185
Cosmos Reason1-7B	Legacy	0.4	0.8	1.05	2.307	1.146
	Structured	0.4	0.8	1.05	2.942	1.517
	One	0.4	0.8	1.0	2.500	1.231

FIGURE 5 – benchmark de lightEMMA sur le dataset nuScenes (10 scènes de 12 frames chacune) qui compare Qwen2.5 VL-7B-Instruct et Cosmos-Reason1-7B avec different systeme de prompt. Source : Effectué par moi-même



FIGURE 6 – Image used.

3.3 Implémentation

Fig 8, 9 et 10 représentent respectivement l'implémentation de P_{def} , P_{ss} et P_{pm} en Jac avec byllm(), tandis que 7 montre l'implémentation de P_{py} en Python classique. La simplicité de l'implémentation en Jac est évidente comparée à la version Python, qui nécessite en plus de mettre en place un parser.

De 8 à 10, on remarque des différences dans le naming des variables et le typage ; nous avons essayé de rendre le code sémantiquement plus explicite pour maximiser la compréhension de byllm().

3.4. Illustration de la formation du prompt par MT-IR et MT-runtime

Selon le papier MTP, un dictionnaire MT-IR est généré a la compilation associant chaque appel à By à son arbre sémantique (11b).

Ici l'arbre est construit recursivement a partir de la signature, des inputs et outputs de la fonction recuperé via l'AST du code source(11a). l'arbre differe legerement si by est utilisé sur une fonction, une méthode ou une classe(11d).

MT-IR est ensuite utilisé pour generer un prompt durant le runtime (11c).

3.4. Résultats

Tableau récapitulatif (moyennes sur 50 essais sur un Apple M5 (10) @ 4.61 GHz avec 16 Go de RAM) :

Métrique	P_{def}	P_{ss}	P_{pm}	P_{py}
mean ADE	5.81	5.54	2.82	2.39
mean FDE	9.13	8.95	5.12	4.57
mean L2 error	5.81	5.54	2.82	2.39
Temps d'inférence moyen (ms)	134.1	114.2	108.1	11.48
Taux d'erreurs de typage (%)	4%	3%	0.8%	0.2%
Lignes de code (LoC)	16	26	28	73

Par exemple, on peut visualiser d'une trajectoire (12) avec un mean ADE de 2.39 m.

FIGURE 7 – Implementation Python

FIGURE 8 – Implementation de P_{def}

FIGURE 9 – Implementation de P_{ss}

FIGURE 10 – Implementation de P_{pm}

Temps d’inférence moyen en ms : 10 fois plus court pour P_{py} que pour les versions avec byllm(). Peut s’expliquer par un biais matériel (byLLM ne supporte peut-être pas l’architecture CUDA).

***Displacement Errors (ADE, FDE, L2)** : P_{py} et P_{pm} ont des performances similaires, bien meilleures que P_{def} et P_{ss} . Même si la philosophie de byllm() n’est pas exactement respectée, il semble que dans ce cas précis, un prompt manuel bien conçu peut considérablement améliorer les performances sans avoir à gérer le parsing de la sortie. P_{def} obtient les moins bonnes performances, mais le fait que ses résultats soient très proches de ceux de P_{ss} qui contient beaucoup d’information additionnelle amène à s’interroger sur le rôle réel des Semstrings dans la synthèse du prompt.

Taille du code (LoC) : de loin la plus petite pour les versions byllm(), montrant la simplicité d’implémentation du paradigme.

Avec byllm(), la frontière entre prompt engineering et “semantic coding engineering” devient parfois floue. On tente de créer des variables et structures de données très spécifiques pour guider la génération du prompt et d’ajouter des indications sémantiques, mais cela ne garantit pas des performances optimales. En effet, pour des tâches complexes comme la prédiction de trajectoire via VLM, la sémantique du code ne semble pas suffisante pour générer un prompt efficace pour un modèle de taille modeste comme qwen2.5VL_8b.

Ou bien mon implémentation en Jac présente certaines coding practices peu adaptées, ce qui a pu empêcher d’exploiter pleinement le potentiel de byllm(). Des essais supplémentaires en variant les conventions de nommage seraient nécessaires pour confirmer ou invalider cette hypothèse.

5. Conclusion et perspectives

Ce projet a permis d'explorer les capacités de `byllm()` pour une tâche complexe de prédiction de trajectoire multimodale. Bien que l'implémentation en Jac soit plus concise, les performances restent inférieures à une approche Python classique avec un prompt.

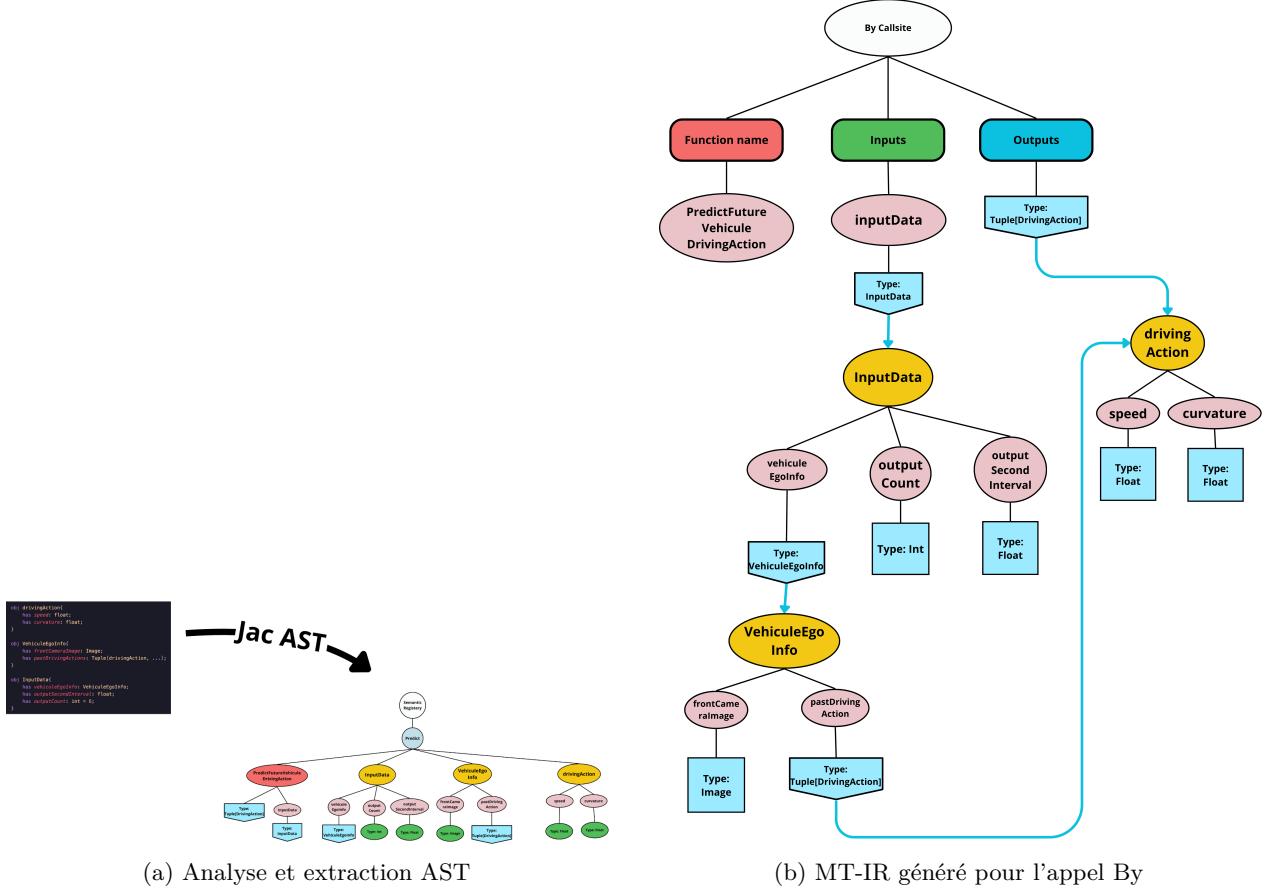
On se rapproche des performances de la version Python uniquement en utilisant un prompt manuel, mais est-ce vraiment dans l'esprit de `byllm()` ?

L'ajout des Semantic String Definitions, qui enrichissent la sémantique du code et n'apparaissent pas dans le papier original, révèle une évolution notable par rapport à l'ambition initiale de s'affranchir du prompt engineering. On se rapproche du concept de DSPy tout en restant flexible.

Malgré cela, il ne faut pas négliger la simplicité d'implementation et la reduction du code nécessaire qui rend son utilisation bien plus agreeable pour des développeurs.

A la lecture des excellents resultats des benchmarks du papier original (avec chatGPT4o), nous nous doutons que des modèles plus puissants GPT-4o pourraient nettement améliorer nos performances actuelles.

Pour les développeurs interagissant régulièrement avec des API d'IA, cette approche constitue une alternative élégante et efficace, et il est probable qu'elle gagne encore en pertinence.



Algorithm 1 MT-IR Constructor Algorithm

Require: Code Base C , Semantic Registry S
Ensure: MT-IR Map M

- 1: Initialize MT-IR $M \leftarrow \emptyset$
- 2: for each by_i call-site in C do
- 3: if by_i is a function-call with name f_i then
- 4: Initialize $M[by_i] \leftarrow \langle f_i, (T_1, \dots, T_n) \rightarrow T_r, m, \theta \rangle$ {From equation 1}
- 5: else if by_i is a method-call mth_i in object of class C_j then
- 6: Initialize $M[by_i] \leftarrow \langle mth_i, C_j, (T_1, \dots, T_n) \rightarrow T_r, m, \theta \rangle$ {From equation 5}
- 7: else if by_i is an object-initialization of class C_j then
- 8: Initialize $M[by_i] \leftarrow \langle C_j, (T_1, \dots, T_k), (T_{k+1}, \dots, T_n), m, \theta \rangle$ {From equation 4}
- 9: end if
- 10: % Extract Parameter Type Semantics:
- 11: for each type T_j in input types of $M[by_i]$ do
- 12: $type_def_j \leftarrow \text{EXTRACTTYPEDEFINITION}(T_j, S)$
- 13: Add $type_def_j$ to $M[by_i]$
- 14: end for
- 15: % Extract Return Type Semantics:
- 16: for each type T_j in output/return types of $M[by_i]$ do
- 17: $type_def_j \leftarrow \text{EXTRACTTYPEDEFINITION}(T_j, S)$
- 18: Add $type_def_j$ to $M[by_i]$
- 19: end for
- 20: end for
- 21: return M

```
[System Prompt]
This is an operation you must perform and return the output values.

[Inputs_Information]
(InputData) (InputData) = Value Bindings

[Output_Information]
(Tuple[drivingAction, ...])

[Type_Explanations]
(InputData)(obj)eg:
- InputData( vehicleEgoInfo=VehiculeEgoInfo, outputSecondInterval = float, outputCount = int)
(VehiculeEgoInfo)(obj)eg:
- VehiculeEgoInfo( frontCameraImage=Image, pastDrivingActions=Tuple[drivingAction, ...])
(drivingAction)(obj)eg:
- drivingAction( speed=float, curvature=float)

[Action]
predictFutureVehicleDrivingAction
Generate and return the output result(s) only, adhering to the provided Type
in the following format
[Output]
<result>
```

(c) Prompt généré par MT-runtime

(d) Algorithme de construction de MT-IR

FIGURE 11 – Étapes successives de la construction et de l'utilisation de MT-IR



FIGURE 12 – Exemple de visualisation des résultats (comparaison des trajectoires prédictes (bleu clair) et réelles (bleu foncé) pour une itération).