PLUSHIE

Programmation en Langage Universel Séquentiel Heuristique pour une Intelligence Évolutive

Quand le calcul émerge de la structure

Programming Language, for a Universal Sequential Heuristic Intelligence Emergence

When computation emerges from structure

Résumé :

PLUSHIE est un langage expérimental minimaliste destiné à explorer l'émergence de la complexité à partir d'éléments extrêmement simples. Il est fondé sur une seule instruction primitive, la copie en double indirection. La mémoire, constituée de cellules de deux adresses devient un graphe implicite adressable. Le langage ne fait pas de différence entre les adresses mémoire, les données, et les programmes exécutables. Les programmes ne sont pas écrits avec une grammaire fixe. Au contraire, le langage apprend progressivement ses instructions pendant la lecture de simples fichiers textes permettant aux structures exécutables d'émerger.

Cette initiative explore l'hypothèse que la complexité calculatoire peut émerger de simples transformations structurelles couplées à l'acquisition dynamique de symboles. Le but à long terme est de démontrer l'universalité computationnelle du langage auto-organisé et de pousser le plus loin possible le minimalisme et la réflexivité.

Ici sont présentés les prérequis architecturaux nécessaires, les mécanismes centraux de PLUSHIE et son modèle d'exécution.

1. Introduction :

La plupart des langages de programmation reposent sur un jeu d'instruction riche, une syntaxe prédéfinie et immuable et un contrôle explicite des structures et procédures.

PLUSHIE expérimente la direction opposée : le minimum d'instruction et d'hypothèses de départ, pas de syntaxe ni de grammaire, une sémantique évolutive et un contrôle minimisé où tout est une donnée modifiable, y compris l'interpréteur lui-même. Le langage tout entier peut se

modifier pendant son exécution. Jusqu'où le calcul peut-il être réduit tout en restant capable d'universalité ?

L'hypothèse centrale est que le calcul complexe peut émerger de transformations topologiques locales appliquées à une mémoire simplifiée et minimale.

Inspiré de *Lisp* et des modèles computationnels fondamentaux (Turing, Lambda Calcul), PLUSHIE mise sur la **c**opie contrôlée, la récursion, et la manipulation de listes chaînées pour exprimer la logique. PLUSHIE explore une nouvelle voie entre machines abstraites, systèmes émergents et programmation expérimentale, où la puissance du calcul ne vient plus de la complexité des instructions mais de la dynamique du système lui-même.

2. Architecture minimale nécessaire :

2.1- Mémoire :

PLUSHIE manipule des données en mémoire. La mémoire a pour seule propriété de pouvoir donner l'adresse d'une nouvelle cellule mémoire disponible, elle doit donc être extensible.

En pratique, la structure nécessaire de la mémoire est un tableau unidimensionnel d'entiers de type Mémoire[x]. Une fois l'adresse mémoire disponible utilisée, le moteur sous-jacent doit pouvoir en générer une autre.

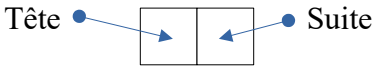
2.2 Données de base :

Les objets de base en PLUSHIE sont des cellules de mémoires constituées d'un couple d'adresses mémoire (T, S). Par analogie avec le LISP, la première donnée est appelée Tête et la seconde Suite.

Par convention : Tête(A) = Mémoire[A]

Dans l'implémentation actuelle, Suite(A) = Mémoire[A+1]

La mémoire est donc un graphe implicite adressable.



Représentation d'une cellule mémoire à deux valeurs

2.3 Entrées de données :

Les informations de l'extérieur sont réduites au minimum : une adresse mémoire particulière reçoit une valeur d'entrée quand elle est vide.

Dans l'implémentation proposée, ces informations sont des caractères qui proviennent d'un fichier texte brut (au format ANSI) : les programmes sont donc juste du texte.

2.4 Sortie de données :

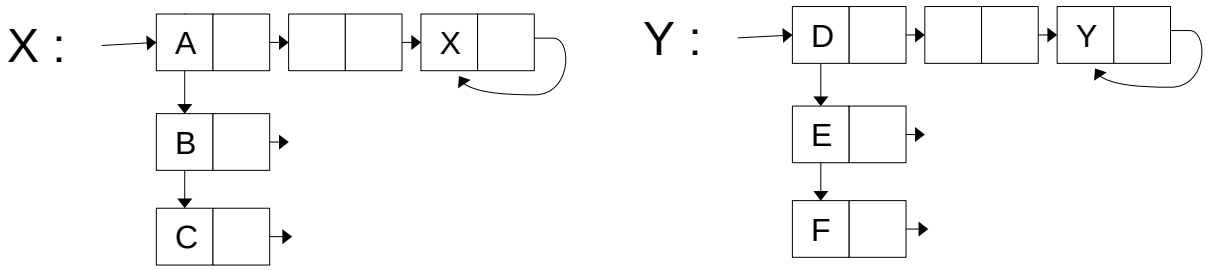
L'interaction extérieure est réduite à l'affichage de symboles sur un écran ou un support quelconque. Dans l'implémentation proposée, il s'agit d'une boite de texte. Une adresse mémoire particulière est écrite dans cette boite de texte si elle est non vide.

3. Instruction Unique :

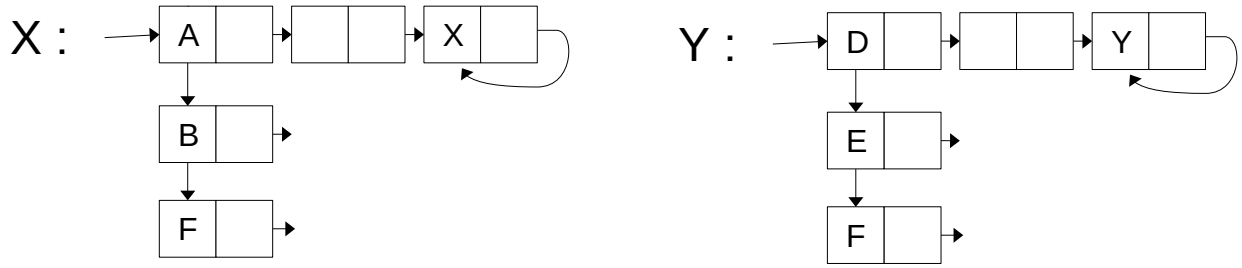
En plus de cette fonction primaire suite, PLUSHIE fonctionne à l'aide la répétition d'une seule instruction, la copie indirecte en double niveau :

Mémoire[Mémoire[Destination]]=Mémoire[Mémoire[Source]]

Situation avant l'exécution de l'instruction unique :



Situation après l'exécution de l'instruction unique : X←Y



Cette instruction permet :
– La redirection de pointeurs
– La construction de structure dynamique
– L'auto transformations
– La simulation de registres, piles, arbres ou graphes symboliques.

Aucune autre instruction n'est nécessaire : pas d'arithmétique, pas de branchement, pas de structure de contrôle prédéfinies.

4. Structure des données :

Les données en mémoire sont organisées en listes chaînées. Une liste chaînée est une série de cellules mémoire dont la suite pointe vers la cellule suivante de la chaîne. Par convention, une liste chaînée se termine quand la suite de la cellule mémoire pointe vers elle-même.

5. Pointeur d’exécution :

Une valeur mémoire spécifique, notée Exec, correspond au pointeur qui adresse la chaîne d’exécution courante.

Chaque pas d’exécution consiste à prendre la première adresse pointée par Exec comme destination et l’adresse suivante comme source, puis à exécuter la copie en double indirection.

La redirection de l’exécution (branchement) s’effectue par l’instruction :

Exec Nouvellechaîne

Comme la chaîne d’exécution est une structure ordinaire de cellules mémoire, PLUSHIE est une machine abstraite réflexive qui peut modifier sa chaîne d’exécution de manière dynamique.

6. Moteur de base :

Une structure, non modifiable durant l’exécution du programme, et réalisable uniquement en hardware si c’est souhaité, se charge de l’exécution d’un moteur fondamental immuable de type automate qui effectue les opérations suivantes : (Exec est la liste chaînée en cours d’exécution)

1- Si la mémoire est saturée, obtenir un nouveau couple d’adresses mémoire.

2- Copier sur le registre Destination la valeur actuelle d’Exec :
Destination = Tete(Tete(Exec))

3- Avancer La chaîne d’exécution :
Tete(Exec)=Tete(Suiv(Tete(Exec))

4- Copier sur le registre Source la valeur actuelle d’Exec :
Source = Tete(Tete(Exec))

5- Avancer La chaîne d’exécution :
Tete(Exec)=Tete(Suiv(Tete(Exec))

6- Effectuer l’instruction unique :
Tete(Tete(Destination))=Tete(Tete(Source))

7- Mettre à jour la valeur qui permet d’obtenir la suite pointée par une adresse donnée :
Tete(Tete(Adressedesuite)) = Suiv(Tete(Tete(Variablesuite)))

8- Si Entrée est vide : lire un caractère

9- Si Sortie est non vide, afficher sortie puis vider sortie.

10- Boucler tant que la fin n’est pas atteinte.

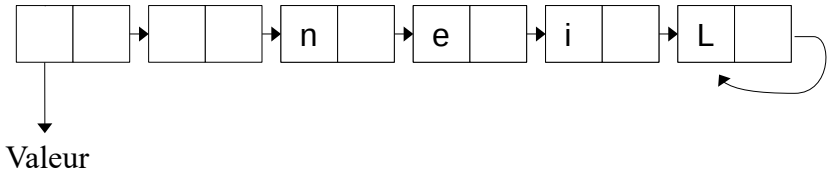
7. Interpréteur initial :

L'interpréteur de base est écrit lui-même en mémoire sous forme d'une liste chaînée d'adresses mémoires interprétables par le moteur de base.

Il ne présuppose aucune grammaire fixe, ni même de séparateur prédéfini. Il dispose au départ d'une chaîne de caractères qui constitue son premier mot connu.

Un mot connu est une liste chaînée dont les têtes des cellules constitutives ont la valeur de code des caractères qui constituent le mot. La première cellule correspond à la valeur du mot connu.

Représentation du mot connu «Lien » en mémoire :



Lorsqu'il isole une liste de caractère entre deux mots connus, l'interpréteur de base enregistre cette liste de caractère comme un nouvel atome nommé connu. Un atome nommé est un mot connu dont la valeur est lui-même, donc sa tête pointe vers sa tête : Mémoire[Atome nommé] = Atome nommé.

Lorsque deux mots connus se succèdent (ce qui inclut une expression qui vient d'être enregistrée comme un nouvel atome nommé), ils forment une instruction exécutable qui est immédiatement exécutée comme la copie en double indirection du second mot sur le premier.

Dans l'implémentation proposée, cet interpréteur de base est contenu dans le fichier plusbase.mem constitué d'environ 300 cellules mémoires (donc 600 adresses au format entier long)

Ce mécanisme permet aux structures symboliques d'émerger progressivement à partir de la donnée d'une chaîne de caractères. Les programmes se compilent en temps réel pendant la lecture.

8. Émergence progressive du calcul :

À partir de cet interpréteur de base, un premier fichier d'initialisation donne un nom aux variables internes suivantes :

– Exécuter : pointeur d'exécution

– Mémoirelibre : adresse de la prochaine cellule mémoire disponible

– Voir : adresse de sortie

– Suitede : Adresse de la variable de la sous fonction suite

– Suite : adresse de la suite de l'adresse contenue dans la variable Suitede.

(en pratique, quand une valeur A est affectée à Suitede, Suite prend la valeur A+1)

Le langage peut alors générer :

– un test d'égalité (saut conditionnel) de type si A égale B alors C sinon D

– des boucles de type tant que A répéter B

– des procédures et fonctions

– un nouvel interpréteur pour lui-même.

Comme les programmes sont eux-mêmes des données en mémoire, des listes chaînées parmi d'autre, ils peuvent s'éditer eux-même en temps réel.

Contrairement aux autres langages, PLUSHIE ne démarre pas avec une notion de nombre prédéfinie. Les nombres émergent naturellement de la structure en tant que longueur des listes chaînées. À partir de là, il est possible de reconstruire toute l'arithmétique.

9. Limitations :

Compte tenu de ses propriétés et de son fonctionnement fortement réflexif, PLUSHIE est difficile à analyser formellement. Il est peu efficace en terme d'exécution et ardu à déboguer du fait des effets indirects à propagation globale. Il ne dispose pas de variables locales. Toute la mémoire est potentiellement impactée par chaque instruction de copie indirecte.

10. Vers l'universalité :

Un objectif clé de recherche est d'implémenter en PLUSHIE une λ-reduction complète.

Puisque

– les structures exécutables sont des graphes.

– les pointeurs peuvent être dupliqués et redirigés

– le contrôle est lui-même une donnée.

PLUSHIE est un bon candidat pour une modèle de calcul universel, complet au sens de Turing.

11. Perspectives :

Les directions de recherche incluent :

– Formaliser le modèle computationnel

– Établir une preuve formelle d'universalité (par exemple implémenter un moteur de réduction de λ-calcul)

– Analyser de l'émergence de la complexité

– Explorer des architectures s'adaptant en temps réel

– Rechercher le noyau minimum à la génération d'une intelligence artificielle fondée sur des graphes réflexifs auto-modifiants.

12. Conclusion :

PLUSHIE explore une hypothèse radicale : le calcul ne nécessite pas d'instructions complexes mais seulement la capacité des structures à se réorganiser d'elles-mêmes.

En combinant une sémantique minimale avec une acquisition symbolique réflexive, il propose une nouvelle approche expérimentale pour un langage simple et la science du calcul.

PLUSHIE appartient à la grande famille des machines minimales à instruction unique, de la réécriture de graphes, des systèmes réflexifs et des langages auto-extensibles. Mais la combinaison de copie indirecte comme instruction primitive, mémoire structurée tête/suite, parsing auto-apprenant sans séparateur générant directement des instructions, d'une chaîne d'exécution auto modifiable et d'un bootstrapping linguistique paraît originale.

Mots clés :

Minimalisme calculatoire, Syntaxe émergente, calcul fondé sur la structure, systèmes réflexifs, langage expérimental.

Annexe :

Exemple concret : le site internet https://pi-plushie.github.io/PLUSHIE-Site/ comporte une interface javascript permettant de tester le langage avec des fichiers de démonstration. Une interface windows y est également disponible au téléchargement.

Exemple d'initialisation :

Une phrase possible de démarrage de PLUSHIE dans le fichier texte d'initialisation peut-être :

Éveil Lien Crée Éveil Ensuite Crée Unmot Ensuite Mémorise Unmot

Lien est le seul mot connu au départ. Il est pointé par le fichier mémoire initial vers l'adresse de Suite, qui pointe elle-même vers l'adresse de Suitede.

Le mot Éveil (choisi ici pour son côté évocateur) est donc crée en tant qu'atome nommé qui pointe vers lui-même.
Ensuite l'instruction Éveil Lien est éxécutée. En conséquence, la valeur d' Éveil est égale à la valeur pointée par la valeur pointée par Lien, c'est-à-dire Suite : Éveil est un synonyme de Suite.

Comme la valeur de suite est écrasée en permanence par l'automate sous jacent, elle peut recevoir n'importe quelle valeur sans affecter la stabilité du programme : elle est donc parfaite pour apprendre de nouveaux atomes nommés sans exécuter de véritable instruction.

Dans l'instruction suivante, Crée Éveil , Éveil est désormais connu, donc Crée est appris et prend comme valeur mémoire(mémoire(Éveil)) c'est-à-dire la valeur pointée par suite (et donc écrasée en permanence par suitede+1). Mais comme au départ, Suitede a une valeur fixe, celle-ci peut être utilisée avant d'affecter une valeur à Suitede.

Dans l'instruction suivante, Ensuite Crée , Ensuite prend la valeur pointée par la tête de suite, qui est configurée en dur dans le fichier de mémoire initial comme l'adresse de la prochaine case de mémoire libre disponible.

Bien entendu les mots n'ont pas de sens pour PLUSHIE. La séquence de départ peut être plus prosaïque :

Suite Lien Déclarer Suite Suitede Déclarer Compteur Suitede Mémoire Compteur

qui produit exactement le même effet.