Théorème de projection idempotente sur schéma arborescent

Auteur : Zakaria Gharzouli **Date** : Juin 2026 **Statut** : résultat à vérifier indépendamment avant publication (Zenodo)

---

0. Contexte et honnêteté sur la portée

Ce document décrit un résultat **plus restreint** qu'une tentative précédente ("Théorème de Gharzouli" sur la résilience par potentiel de désordre), qui s'est révélée être une instance correcte mais non originale du théorème du point fixe de Banach (1922), une fois sa métrique et ses opérateurs corrigés.

Ce nouveau résultat répond à une question différente et plus modeste :

> *Étant donné un schéma de référence `T*` et un état observé `T`, peut-on*
>
> ***projeter T sur la conformité à ce schéma en une seule opération**,*
>
> *sans itération, sans notion de distance, sans suite qui converge ?*

La réponse est oui, et le mécanisme de preuve (identité algébrique d'idempotence `P∘P = P`) est structurellement différent de Banach : il ne repose sur aucune notion de distance décroissante, de compteur, ou d'ordre partiel itéré. C'est une preuve **en une étape**, pas une preuve de limite.

Ce que ce théorème NE fait PAS : il ne garantit pas la convergence vers les *valeurs exactes* de `T*`. Il garantit la conformité au *schéma* de `T*` (types, bornes, structure), en préservant toute valeur observée qui est déjà conforme — même si elle diffère de `T*`. C'est un théorème de **validation/normalisation instantanée**, pas de **réparation exacte vers une référence ponctuelle**. Toute présentation de ce résultat doit énoncer cette limite explicitement.

---

1. Définitions

1.1 Espace des arbres étiquetés à conteneurs explicites

Définition 1.1. Un arbre étiqueté est un triplet `T = (N, κ, V)` où :

- `N` est un ensemble fini de nœuds, organisés en arborescence par labels.

- `κ : N → {leaf, dict, list}` est une fonction de type de nœud, *explicite*

et indépendante du nombre d'enfants (en particulier, un nœud `dict` ou `list` avec zéro enfant reste un conteneur, jamais une feuille — cette distinction est nécessaire et a fait défaut dans une version antérieure du travail, où `{}` était confondu avec une feuille de valeur `None`).

- `V : N_{leaf} → 𝒱` est une fonction de valeur sur les feuilles, avec

`𝒱 = ℚ ∪ String ∪ {None} ∪ {NaN, ±Inf}`.

1.2 Étiquette de type

Définition 1.2. Pour `v ∈ 𝒱`, l'étiquette de type `τ(v)` est :

- `τ(v) = 'none'` si `v = None`
- `τ(v) = 'invalid'` si `v ∈ {NaN, +Inf, -Inf}`
- `τ(v) = 'number'` si `v` est un nombre fini (entier ou flottant, hors

`bool`, qui est traité comme un type distinct pour éviter la coercition `True == 1` de certains langages)

- `τ(v) = type(v)` sinon (ex. `'str'`)

1.3 Schéma de bornes

Définition 1.3. Un schéma de bornes est une fonction partielle `β : Paths(T*) ⇀ ℚ × ℚ` qui associe à certains chemins de feuilles numériques de `T*` un intervalle `[lo, hi]`.

1.4 Conformité

Définition 1.4. Une valeur `v` au chemin `p` est **conforme** au schéma de `T*` si :

1. `τ(v) = τ(v*(p))` où `v*(p)` est la valeur de `T*` au chemin `p`, **et**
2. si `p ∈ dom(β)` et `τ(v) = 'number'`, alors `v ∈ [β(p).lo, β(p).hi]`.

1.5 Opérateur de projection sur schéma

Définition 1.5. L'opérateur `P(T, T*, β)` est défini récursivement sur la structure de `T*` (référentiel fixe) :

- Si `T*` est une feuille au chemin `p` :

`` `P(T, T*, β)(p) = v(p) si v(p) est conforme au schéma en p = v*(p) sinon ` `` où v(p) `est la valeur de` T `au chemin` p `(ou` None `si` T` ne possède pas ce chemin, ou si le nœud présent n'est pas une feuille).

- Si `T*` est un conteneur `dict` au chemin `p` :

`P` construit un nouveau conteneur `dict` avec, pour chaque enfant `c*` de `T*`, l'enfant `P(c_T, c*, β)` où `c_T` est l'enfant de même label dans `T` (ou `None` si absent). Les enfants de `T` sans label correspondant dans `T*` sont **ignorés** (supprimés du résultat).

- Si `T*` est un conteneur `list` de longueur `n` au chemin `p` :

Si `T` possède aussi un conteneur `list` de longueur `n` au même chemin, on choisit d'abord, parmi les `n` rotations cycliques de cette liste, celle qui maximise le nombre de positions immédiatement conformes (ties : la rotation nulle, c'est-à-dire l'identité, est préférée). On applique ensuite `P` récursivement position par position. Si `T` ne possède pas de liste de même longueur, chaque position est traitée comme si l'enfant de `T` à cette position était absent.

---

2. Théorème

Théorème (Idempotence de la projection sur schéma).

Soit `T*` un arbre de référence et `β` un schéma de bornes tel que `T*` soit lui-même conforme à `β` en tout point de son domaine (précondition de cohérence : `∀p ∈ dom(β), v*(p) ∈ [β(p).lo, β(p).hi]`). Alors pour tout arbre observé `T` :

```
P(P(T, T*, β), T*, β) = P(T, T*, β)
```

C'est-à-dire que `P(·, T*, β)` est une projection idempotente sur l'espace des arbres conformes au schéma de `T*`. De plus, ceci se vérifie en une seule application de `P`, sans itération, sans notion de distance, et sans borne de "nombre d'étapes" puisqu'aucune suite n'est impliquée.

2.1 Preuve

Par récurrence structurelle sur `T*`.

Cas de base (feuille). Soit `p` un chemin de feuille de `T*`, de valeur de référence `v*(p)`, et soit `v = P(T, T*, β)(p)`. Deux cas se présentent selon la Définition 1.5 :

- *Cas A* : `v(p)` (valeur originale dans `T`) était conforme, donc

`v = v(p)`. Comme `v` est conforme par hypothèse de ce cas (même type, bornes respectées si applicable), en réappliquant `P`, la valeur en entrée à cette étape est `v`, qui est conforme — donc `P(P(T,T*,β), T*, β)(p) = v = P(T,T*,β)(p)`. ✓.

- *Cas B* : `v(p)` n'était pas conforme, donc `v = P(T,T*,β)(p) = v*(p)`

(la valeur de référence elle-même). Par la **précondition de cohérence**, `v*(p)` est conforme à son propre schéma (même type que lui-même trivialement, et dans ses propres bornes par hypothèse).

Donc en réappliquant `P` à l'entrée `v*(p)`, on est dans le Cas A avec `v(p) = v*(p)` conforme, ce qui donne `P(P(T,T*,β),T*,β)(p) = v*(p) = P(T,T*,β)(p)`. ✓.

Cas inductif (dict). Par hypothèse de récurrence, chaque enfant `c*` de `T*` vérifie `P(P(c_T,c*,β),c*,β) = P(c_T,c*,β)`. Comme `P` sur un nœud `dict` ne fait qu'assembler les résultats récursifs sur chaque enfant de `T*` (sans transformation additionnelle au niveau du nœud lui-même), l'idempotence du nœud `dict` découle directement de celle de ses enfants. ■

Cas inductif (list). Soit `L = P(T,T*,β)` au nœud liste considéré. Par construction, `L` a la même longueur que `T*` (les positions manquantes sont traitées comme absentes, donc remplies par les valeurs de référence — cas B ci-dessus — qui sont conformes par cohérence). Lorsque `P` est réappliqué à `L` :

1. `L` a déjà la bonne longueur, donc `best_list_alignment` est appelé sur

une liste qui est déjà dans une configuration optimale ou équivalente (la rotation identité reste candidate et n'est jamais pire qu'une autre rotation pour une liste déjà conforme position par position, par construction du score de conformité, qui est maximal une fois que toutes les positions sont conformes — un maximum ne peut être dépassé).

1. Chaque position de `L` est conforme (par le cas de base, déjà établi),

donc chaque position est inchangée par la seconde application de `P`.

Donc `P(L, T*, β) = L`. ■

Précision sur le choix de rotation en cas d'égalité (tie-break). La sélection de rotation utilise une comparaison strictement supérieure (`score_candidat > meilleur_score_courant`), et teste la rotation identité en premier. En cas d'égalité de score entre plusieurs rotations, la rotation identité (ou, plus généralement, la première rotation testée atteignant le score maximal) est donc toujours conservée de façon déterministe. C'est un choix de tie-break explicite et stable, nécessaire à la preuve d'idempotence ci-dessus : sans cette stabilité, deux applications successives de `P` pourraient en principe sélectionner deux rotations différentes de même score, ce qui casserait l'idempotence sur les nœuds de type `list`. Ce point a été vérifié indépendamment (juin 2026) sur des cas construits spécifiquement pour avoir plusieurs rotations de score égal ; le tie-break déterministe tient dans tous les cas testés.

Conclusion. Par récurrence structurelle sur `T*` (fini par hypothèse, Définition 1.1), `P(P(T,T*,β),T*,β) = P(T,T*,β)` pour tout `T`. ■

2.2 Remarque sur la nature de la preuve

Cette preuve ne fait intervenir à aucun moment :

- une métrique ou une notion de distance entre arbres,
- une suite `(T_n)` et une limite,
- un argument de descente combinatoire (compteur qui décroît),
- un ordre partiel itéré.

Elle repose uniquement sur une **disjonction de cas finie au niveau de chaque feuille**, combinée par récurrence structurelle sur un arbre fini.

Précision suite à vérification indépendante : dire que cette preuve est « structurellement différente de Banach » est correct mais peut être formulé plus précisément. La preuve n'utilise pas le cadre métrique de Banach (espace métrique complet, opérateur contractant, suite itérée vers un point fixe unique) : elle s'apparente plutôt à la démonstration de l'idempotence d'un projecteur algébrique (`P² = P`), un objet connu en algèbre linéaire depuis le début du XXe siècle (matrices de projection, `P = X(XᵗX)⁻¹Xᵗ` en régression linéaire), mais ici construit explicitement sur un espace d'arbres étiquetés avec des règles de conformité de type/bornes et préservation sélective. La différence avec Banach est donc réelle (mécanisme de preuve différent, point fixe non unique — tout arbre conforme au schéma est un point fixe de `P`, alors que Banach garantit l'unicité), mais la brique de base (l'idempotence comme propriété algébrique) n'est pas elle-même nouvelle ; c'est son application à cette structure de données précise qui ne semble pas formalisée ainsi dans la littérature existante (voir section 4, vérification de nouveauté).

---

3. Portée exacte et limites (à ne jamais omettre)

3.1 Ce que le théorème garantit

- `P` ramène tout arbre `T` à un arbre structurellement conforme au schéma

de `T*` (types corrects, bornes respectées, structure alignée) en **une seule application**, déterministe, sans itération.

- `P` est idempotent : ré-appliquer `P` à un résultat déjà conforme ne le

change pas.

- `P` préserve toute valeur de `T` qui est déjà conforme, **même si elle

diffère de la valeur exacte de `T*`**.

3.2 Ce que le théorème NE garantit PAS

- **Aucune garantie de convergence vers les valeurs exactes de `T*`.** Une

corruption qui change une valeur numérique en restant dans le même type et les mêmes bornes (ex. `50` corrompu en `999` sans bornes déclarées, ou `50` corrompu en `80` avec des bornes `[0,100]`) **n'est pas détectée ni corrigée**. C'est la limite fondamentale par rapport à un objectif de "réparation exacte" — ce théorème répond à une question de conformité structurelle, pas d'égalité ponctuelle.

- **La précondition de cohérence est obligatoire.** Si `T*` lui-même viole

les bornes `β` qu'on lui associe, le théorème ne fait aucune promesse (l'idempotence peut tout de même tenir empiriquement dans certains cas, mais ce n'est pas couvert par la preuve).

- **Pas de gestion de graphes cycliques ou de DAG** : le théorème est

énoncé pour des arbres finis stricts (Définition 1.1).

- **Les bornes sont indépendantes entre elles** : aucune contrainte

croisée entre deux chemins n'est exprimable dans ce cadre (pas de "si `a` > 10 alors `b` doit être positif").

3.3 Limites supplémentaires identifiées par vérification indépendante (juin 2026)

- **Limite de récursion Python.** L'implémentation de référence est

récursive (une frame par niveau structurel, plusieurs frames Python par niveau logique de l'arbre). Avec la limite de récursion par défaut de Python (1000), la profondeur maximale supportée est d'environ **990** niveaux, pas 1500 ou 3000 comme indiqué dans une vérification préliminaire — ce chiffre a été mesuré directement et corrigé. Pour des arbres plus profonds, l'appelant doit augmenter explicitement la limite via `sys.setrecursionlimit(...)`, comme documenté dans `final_validation.py`. Ceci n'affecte pas la validité du théorème (énoncé pour des arbres finis, sans bornage de profondeur), seulement les limites pratiques de cette implémentation Python particulière.

- **Comparaison de valeurs `NaN`.** Toute comparaison d'égalité entre deux

arbres résultats (par exemple pour vérifier empiriquement l'idempotence) doit utiliser une égalité structurelle qui traite deux `NaN` comme égaux (fournie : `safe_eq`), et non l'opérateur `==` natif de Python, qui renvoie `False` pour deux objets `NaN` distincts même quand ils représentent la même valeur "invalide" au sens du schéma. Cette distinction n'affecte pas l'opérateur `P` lui-même (qui ne compare jamais deux `NaN` entre eux dans sa logique de réparation), mais affecte toute vérification empirique de l'idempotence basée sur `==` brut, qui peut alors signaler de faux échecs (ou, par un effet d'implémentation Python sans rapport avec la preuve, masquer artificiellement un cas problématique si les deux résultats comparés réutilisent par hasard le même objet flottant en mémoire). Ce point a été identifié lors d'une vérification indépendante et corrigé dans `schema_idempotent_engine.py`.

- **Clés de dictionnaire hétérogènes.** Si l'objet Python d'entrée contient

des clés de types différents mais de même représentation textuelle (par exemple la clé entière `1` et la

clé chaîne `'1'` dans le même dictionnaire — un cas marginal et rare en pratique), l'ordre interne de traitement de ces deux clés suit l'ordre d'insertion du dictionnaire source plutôt qu'un ordre canonique explicite. Il a été vérifié que cela n'affecte ni la correction du résultat final ni l'idempotence (l'égalité de dictionnaire en Python ne dépend pas de l'ordre des clés) ; c'est une particularité d'implémentation sans conséquence pratique observée.

- **`bool` est traité comme un type distinct de `number`**, par choix de

conception explicite (pour éviter la coercition `True == 1` du langage hôte) : une valeur de référence booléenne ne sera jamais jugée conforme par une valeur entière de même valeur logique, et réciproquement.

---

4. Vérification de nouveauté effectuée

Une recherche a été conduite (juin 2026) sur les notions de projection idempotente, schéma, et réparation arborescente. Les résultats trouvés portent sur :

- l'idempotence de matrices de projection en algèbre linéaire et en

statistique (régression linéaire, $P = X(X^tX)^{-1}X^t$), connue depuis le début du XXe siècle ;

- les semi-modules idempotents (algèbre max-plus) ;
- les quasi-groupes idempotents en algèbre abstraite ;
- une formalisation académique de JSON Schema (publiée à POPL'24,

arXiv:2307.10034) qui prouve que le problème de **validation** d'une donnée contre un schéma JSON est PSPACE-complet. Ce travail porte sur la décision booléenne "cette donnée est-elle valide ?", pas sur la **réparation** ou la **projection idempotente** d'une donnée invalide vers une donnée valide en préservant le maximum d'information — c'est un travail connexe mais répondant à une question différente de celle traitée ici. Trouvé et confirmé lors d'une vérification indépendante.

Aucune formalisation trouvée ne correspond à l'opérateur `P` défini en Section 1.5 (projection idempotente sur schéma arborescent avec préservation sélective par conformité de type/bornes). La brique de base (l'idempotence comme propriété) est ancienne et bien connue ; la construction spécifique présentée ici ne l'est pas, à la connaissance de l'auteur et après la recherche effectuée (incluant une vérification indépendante par un second modèle) — **mais cette vérification de nouveauté n'est pas exhaustive et doit être complétée avant toute publication**, notamment du côté de la littérature en réécriture de termes confluente (formes normales, lemme de Newman) et en réparation automatique de données structurées (auto-repair, data healing), domaines dans lesquels des constructions voisines pourraient exister sous une autre terminologie.

5. Validation empirique

5.1 Tests internes initiaux

- 0 échec d'idempotence sur plus de 13 000 cas générés aléatoirement

(arbres de profondeur 3–4, taux de corruption 0.1 à 0.95, incluant changements de type, valeurs `None`, `NaN`, `Inf`).

- Cas limites validés : dict vide, T* égal à l'entrée, structures

totalement disjointes, bornes incohérentes, rotation de listes.

5.2 Vérification indépendante (juin 2026)

Une vérification adversariale par un second modèle (Kimi) a été conduite sur ce travail, avec instruction explicite de chercher à le réfuter. Elle a permis d'identifier :

- une comparaison d'égalité non NaN-safe dans les tests internes

(corrigée, voir `safe_eq` dans `schema_idempotent_engine.py`),

- une affirmation erronée sur la limite de récursion Python par défaut

dans une vérification préliminaire (corrigée : la limite réelle est d'environ 990 niveaux, pas 1500),

- l'observation sur les clés hétérogènes et le tie-break de rotation

documentées en Section 3.3,

- la référence JSON Schema (POPL'24) intégrée en Section 4.

Aucun contre-exemple réfutant l'idempotence elle-même n'a été trouvé après cette vérification.

5.3 Validation finale post-corrections

- 10 000 cas supplémentaires testés après corrections, avec comparaison

NaN-safe : 0 échec.

- Tous les scénarios adversariaux proposés par la vérification

indépendante rejoués explicitement (bool/int, clés hétérogènes, tie-break de rotation, listes de longueurs différentes, précondition de bornes violée, récursion profonde avec la limite réelle mesurée) : tous passent.

- Performance : 100 000 clés réparées en environ 0.6 s (une seule passe,

pas d'itération).

Voir `final_validation.py` pour le détail complet de ces tests, et `schema_idempotent_engine.py` pour l'implémentation de référence.