RAPPORT DE PROJET FONCTIONNEL
– *Projet pieuvre* –

Hugo Salou et Thibaut Blanc

27 mai 2025

Table des matières

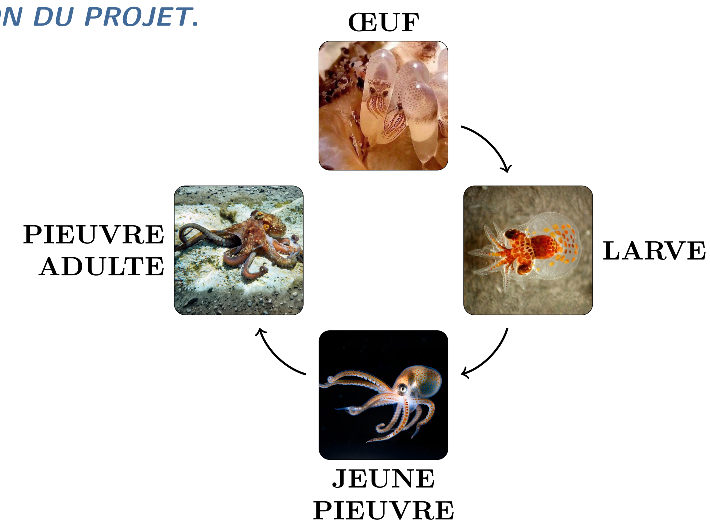

Figure 1 | *Cycle de vie de la pieuvre*

1 L'ŒUF : *MANIPULATION DES TERMES.*

Au cœur de notre pieuvre, nous avons une syntaxe simple représentant à la fois les termes et les types. On note $\mathscr{V}$ un ensemble infini de variables. La grammaire définissant la syntaxe est :

$$M, N ::= x \mid M\ N \mid \lambda(x : M).\ N \mid \Pi(x : M).\ N \\ \mid \mathsf{Type}_i \mid \mathsf{Set} \mid \mathsf{Prop},$$

où $x \in \mathscr{V}$ et $i \in \mathbb{N}$.

Les deux premières constructions sont identiques à celles équivalentes pour les λ-termes. Pour la λ-abstraction, on donne explicitement le type de x qui est, comme dit plus tôt, aussi un terme. On contient donc une partie de la syntaxe recommandée dans [?] au niveau des termes.

Les Π-types sont les types des λ-abstractions : ainsi, le terme $\Pi(x : M).\ N$ est une généralisation du type $M \to N$ où le type N peut dépendre de la valeur de x choisie. Lorsque $x \notin \mathscr{V}\ell(N)$, on retrouve le cas *non-dépendant*, c'est-à-dire le cas du type flèche $M \to N$. D'un point de vue « logique », on unifie l'implication $\Rightarrow$ et le quantificateur universel $\forall$.

Finalement, il reste les *sortes* : Type_i, Set et Prop. L'idée est que si l'on avait un unique type Type des types, alors on aurait un système incohérent (*c.f.* paradoxe de GIRARD). À la place, le type d'un Π-type est une sorte, et le type d'une sorte est une sorte « d'un niveau juste au dessus », ce qui empêche d'avoir $t : t$ pour un certain terme t.

1.1 Représentation en pieuvre des termes.

En pieuvre, les termes sont représentés exactement comme dans la grammaire, à quelques différences près. Pour représenter une variable $x \in \mathscr{V}$, on n'utilise pas une chaîne de caractères mais un entier. Ainsi, gérer le *shadowing* se fait simplement : lorsqu'on redéfinit x, on choisit un identifiant différent pour la suite. La génération de variables fraîches se fait aussi très simplement avec un compteur.

Pour faire le lien entre « chaîne de caractère donnée par l'utilisateur » et « identifiant de variable interne », on maintient deux structures de données : un arbre PATRICIA [?] pour le lien « `str` → `id` » et un arbre rouge–noir pour le lien « `id` → `str` ». Ces deux structures de données optimisées permettent de réduire le temps nécessaire pour le confort de l'utilisateur.

Une autre possibilité aurait pu être les *indices de* DE BRUIJN, qui élimine les problèmes liés à l'α-conversion et la capture de variables libres. C'est cette option que l'assistant **Rocq** a choisi.

1.2 La β-réduction et l'α-équivalence.

La construction « Π-type » est aussi un lieur, comme la λ-abstraction. Ainsi, au moment de tester l'α-équivalence, il faut traîter ce cas de manière quasi-identique à une λ-abstraction. Quant à la β-réduction, il suffit d'implémenter la règle :

$$(\Pi(x : M).\ N)\ K \quad \longrightarrow_\beta \quad N[K/x].$$

Pour tester l'$\alpha\beta$-équivalence entre M et N, on commence par β-normaliser M en $M^\star$ et N en $N^\star$, et ensuite on teste si $M^\star =_\alpha N^\star$. On peut se permettre de β-normaliser car tout terme *typé* est *fortement normalisant.*

1.3 Inférence des types et coercition.

Pour la partie « *type checking/type inference* » de notre pieuvre, nous avons deux mécanismes qui fonctionnent en simultané : l'*inférence* et la *vérification.* Un cas important est celui de l'*application* : on souhaite inférer le type de $M\ N$. Pour cela, voici comment nous procédons :

- ▷ commencer par inférer le type de M ;
- ▷ le type de M doit nécessairement être de la forme $\Pi(x : A).\ B$ pour faire l'application ;
- ▷ ensuite vérifier que le type de N est bien « compatible » avec A ;
- ▷ en déduire que $M\ N : B[N/x]$.

Le mécanisme « est compatible » s'appelle la *coercition* : il définit une relation de sous-typage $\preceq$ et où les termes sont égaux modulo $\alpha\beta$-équivalence.

L'utilisateur de notre pieuvre n'indique pas explicitement le « niveau » i de Type_i, mais écrit simplement `Type` en laissant à pieuvre le choix du niveau. Pour réaliser cela, lorsqu'on écrit `Type`, on lui associe un sommet v dans un certain graphe. Le mécanisme de coercition correspond à ajouter des arêtes dans ce graphe et s'assurer qu'il ne contienne pas de cycle. Cette détection se fait à l'aide des procédures données dans [?].

2 LA LARVE : *CALCUL DES CONSTRUCTIONS INDUCTIVES.*

La partie « calcul des constructions inductives » se base principalement sur l'article [?]. On la décompose en trois sous-sections :

1. les définitions inductives avec `Inductive` ;
2. déconstruire un terme inductif avec `match` ;
3. fonction récursive avec `fix`/`Fixpoint`.

2.1 Définitions inductives.

Une définition inductive se décompose en plusieurs parties : le nom, les paramètres, l'arité, et les noms des constructeurs et leur type.

```
Inductive ⟨nom⟩ ⟨paramètres⟩ : ⟨arité⟩ :=
| ⟨nom constructeur 1⟩ : ⟨type constructeur 1⟩
⋮                         ⋮
| ⟨nom constructeur n⟩ : ⟨type constructeur n⟩.
```

Code 1 | *Syntaxe d'une définition inductive*

Plusieurs vérifications sont nécessaires pour vérifier que la définition est *valide.* Par exemple, on doit vérifier que les occurrences du type inductif dans les types des constructeurs n'apparaissent que positivement. Ces instances du type inductif peuvent apparaître dans un autre type inductif mais seulement si celui-ci apparaît dans les paramètres.

Une fois ces vérifications terminées, on ajoute le type inductif et les constructeurs dans l'environnement global.

2.2 Filtrage avec `match`.

```
match ⟨expression⟩ as ⟨alias⟩
in ⟨pattern inductif de l'expression⟩
return ⟨type de retour⟩
with
| ⟨motif 1⟩ ⇒ ⟨expression 1⟩
⋮           ⋮
| ⟨motif n⟩ ⇒ ⟨expression n⟩
end
```

Code 2 | *Syntaxe d'un filtrage dépendant*

Le filtrage permet « d'extraire » les données dans les types inductifs. Ici, le `match` est dépendant (d'où l'apparition de tous les champs) et n'est pas complexe (on ne déconstruit qu'une étape à la fois, et chaque branche doit être présente). Le « n'est pas complexe » peut sembler restreignant mais tout `match` plus avancé peut se traduire en (potentiellement) plusieurs `match`s imbriqués.

2.3 Récursion avec `fix`.

Avec un `fix`, on définit une fonction qui s'appelle sur elle-même *mais uniquement* sur des valeurs structurellement plus petites. Cette comparaison a lieu uniquement sur le dernier argument de la fonction. On doit réaliser une telle vérification pour ne pas que le code 2.3 soit valide (qui induirait un système non-consistant).

```
Definition f (A B : Type) : A → B :=
  fix g (x : A) : B := g x.
```

Code 3 | *Un code pieuvre (et **Rocq**) invalide*

Lorsqu'on écrit `Fixpoint f ... {struct` x_k`}`, on traduit ceci en une définition (avec `Definition`) d'un `fix` qui prend les k premiers arguments et renvoie une fonction prenant les $n-k$ autres arguments.

3 LA JEUNE PIEUVRE : *TACTIQUES SIMPLES*.

Pour implémenter les tactiques, on utilise une méthode « par continuations » pour créer un terme sans avoir à introduire de trous dans la grammaire des termes. Un *problème* est un contexte représentant les hypothèses, suivi d'un objectif. Une *tactique* est une fonction transformant un problème $\mathscr{P}$ en une liste de problèmes $\mathbb{Q}_1, \dots, \mathbb{Q}_n$ (les sous-problèmes suivants à traîter) et une autre fonction qui prend les termes « solutions » des sous-problèmes $\mathbb{Q}_1, \dots, \mathbb{Q}_n$ et qui reconstruit le terme solution de $\mathscr{P}$.

Notre pieuvre supporte également l'utilisation de `..;..`, de `[..|..|..]`, de `fail`, de `do`, de `try`, de `repeat` et de `first`.

3.1 Les tactiques `intro` et `intros`.

La tactique `intro` permet d'introduire une variable. Elle réalise l'opération suivante :

$$[\,??:\Pi(x:A).\,B]\xrightarrow[x\text{ frais}]{\texttt{intro }x}\lambda(x:A).\,[\,??:B]_1.$$

On peut potentiellement α-renommer le x en un autre nom donné par l'utilisateur.

La tactique `intros` correspond à appliquer `intro` encore et encore jusqu'à ce que l'on ne puisse plus (ou que l'on épuise les noms de variables donnés).

3.2 Les tactiques `exact` et `assumption`.

La tactique `exact` permet d'appliquer une hypothèse, ou de donner directement un λ-terme qui vérifie le but. Elle réalise l'opération suivante :

$$[\,??:A]\xrightarrow[M:A]{\texttt{exact }M}M.$$

La tactique `assumption` essaie de faire `exact` H pour toute hypothèse H. Elle plante si aucune hypothèse ne correspond.

3.3 Les tactiques `cut` et `assert`.

La tactique `cut` introduit une coupure (qui sera directement retirée lorsqu'on écrira `Qed` car on normalise le terme crée). Elle procède ainsi :

$$[\,??:A]\xrightarrow{\texttt{cut }B}[\,??:B\to A]_2\,[\,??:B]_1.$$

La tactique `assert` est très similaire, elle introduit juste l'hypothèse dans le 2nd sous-problème :

$$[\,??:A]\xrightarrow[x\text{ frais}]{\texttt{assert }B}(\lambda(x:B).\,[\,??:A]_2)\,[\,??:B]_1.$$

3.4 La tactique `set`.

On définit une variable pour avoir une certaine valeur. On ajoute donc x:=M aux hypothèses. On réalise :

$$[\,??:A]\xrightarrow[M:B]{\texttt{set }x\texttt{ := }M}(\lambda(x:B).\,[\,??:A]_1)\,M.$$

3.5 La tactique `pattern`.

La tactique `pattern` « extrait » une variable de l'objectif :

$$[\,??:A]\xrightarrow[y\text{ frais},x:B]{\texttt{pattern }x}\left[\,??:(\lambda(y:B).A[x/y])\,x\right]_1.$$

Elle peut sembler « peu utile » mais elle prendra son utilité dans la sous-section 4.3.

3.6 Les tactiques `unfold`, `compute`, `simpl`, et `change`.

Ces trois tactiques ne modifient pas le terme généré mais modifie le but.

- Dans le cas de `unfold`, on réalise une δ-réduction (*i.e.* on remplace une variable par sa définition partout dans le terme).
- Dans le cas `compute`, on $\beta\delta\iota$-normalise le but (la ι-réduction permet de gérer les `match` et les `fix`).
- Dans le cas de `simpl`, on réalise une β-réduction.
- Dans le cas de `change`, on change le but en un autre qui est $\alpha\beta\delta\iota$-équivalent.

3.7 La tactique admit.

Lorsqu'on ne sait pas montrer A, on applique admit qui rajoute dans l'environnement global une variable ayant pour type A, et on place cette variable dans le terme :

$$[\,??:A]\xrightarrow[\substack{\Gamma\to\Gamma,x:A\\ x \text{ frais}}]{\texttt{admit}} x.$$

Attention cependant, il devient alors impossible de conclure la preuve avec Qed, il faut utiliser Admitted signifiant que cette preuve n'est complète. Il sera néamoins possible d'utiliser ce théorème dans d'autres preuves, comme n'importe qu'elle autre théorème.

3.8 Les tactiques clear et idtac.

Pour clear, on ne modifie pas le terme, on retire juste une hypothèse x de la liste des hypothèses. On doit vérifier que x n'apparait pas dans une autre hypothèse. La tactique idtac ne fait absolument rien.

4 LA PIEUVRE ADULTE : *TACTIQUES PLUS COMPLEXES.*

4.1 La tactique apply.

La tactique apply réalise l'opération suivante :

$$\begin{array}{c}[\,??:A]\\ M:\Pi(x_1:B_1)\ldots(x_n:B_n).A \;\Big\downarrow\; \texttt{apply}\ M\\ M\,[\,??:B_1]_1\cdots[\,??:B_n]_n.\end{array}$$

On engendre ainsi n sous-buts en exploitant l'hypothèse (ou un terme) M.

4.2 Les tactiques left, right, split et reflexivity.

Dans chaque cas, on applique le constructeur correspondant à la tactique voulue avec apply. On obtient donc :

$$\begin{aligned}[\,??:A\vee B] &\xrightarrow{\texttt{left}} \texttt{or\_introl}\ A\ B\,[\,??:A]_1\\ [\,??:A\vee B] &\xrightarrow{\texttt{right}} \texttt{or\_intror}\ A\ B\,[\,??:B]_1\\ [\,??:A\wedge B] &\xrightarrow{\texttt{split}} \texttt{conj}\ A\ B\,[\,??:A]_1\,[\,??:B]_2\\ [\,??:M=M] &\xrightarrow[M:A]{\texttt{reflexivity}} \texttt{eq\_refl}\ A\ M.\end{aligned}$$

4.3 Les tactiques induction et elim.

Lorsqu'on a induction H, on commence par introduire toutes les variables jusqu'à H, puis on applique le principe inductif correspondant (c'est un assemblage fix et match).[1] On termine avec un simpl et un clear de la variable H temporaire.

La tactique elim réalise la même chose que induction mais sans intros, ni clear.

4.4 Les tactiques exfalso et absurd.

On utilise pour ces tactique le principe inductif de False.

```
False_ind
   : forall P : False → Prop, forall a : False, P a
```

Code 4 | *Principe inductif de* False*.*

La tactique exfalso procède ainsi :

$$[\,??:A]\xrightarrow{\texttt{exfalso}} \texttt{False\_ind}\ A\,[\,??:\texttt{False}]_1.$$

La tactique absurd fait comme exfalso mais ajoute une coupure :

$$[\,??:A]\xrightarrow{\texttt{absurd}\ B} \texttt{False\_ind}\ A\,([\,??:\neg B]_2\,[\,??:B]_1),$$

où la notation $\neg B$ signifie $B \to \texttt{False}$.

4.5 Les tactiques rewrite et rewrite_.

Les tactiques rewrite et rewrite_ (c'est noté rewrite ← en **Rocq**) appliquent le principe inductif de l'égalité.

Pour rewrite_ H où $H : M = x$ dans les hypothèses, l'idée est que l'on réalise pattern pour extraire le x de l'objectif, puis on applique le principe inductif de l'égalité ce qui réalise une substitution dans l'objectif.

Pour la tactique rewrite, on utilise un lemme puissant : l'égalité est symétrique. Il suffit donc de prendre le symétrique de l'hypothèse donnée, et on n'a plus qu'à appeler rewrite_ (classique) directement.

4.6 La tactique injection.

La tactique injection s'applique lorsqu'on a une égalité de deux constructeurs en hypothèse. On veut montrer que l'on a les mêmes arguments. Pour cela, procédons sur l'exemple de $\texttt{S}\ n = \texttt{S}\ m \to n = m$. On applique le principe inductif de l'égalité pour montrer :

$$\text{si } \texttt{S}\ n = d \text{ alors } n = \begin{cases} n & \text{si } d = \texttt{O}\\ k & \text{si } d = \texttt{S}\ k,\end{cases}$$

qu'on applique dans le cas où $d = \texttt{S}\ m$, ce qui donnera $n = m$, comme demandé.

4.7 La tactique discriminate.

La tactique discriminate permet d'éliminer les cas où l'on a une hypothèse de la forme $\texttt{C}_1\ \ldots = \texttt{C}_2\ \ldots$ avec $\texttt{C}_1$ et $\texttt{C}_2$ différents. Comme pour injection, on montre sur un exemple. Montrons $\texttt{O} = \texttt{S}\ n \to A$. On applique exfalso pour montrer que si $\texttt{O} = \texttt{S}\ n$ alors False. On applique le principe inductif de l'égalité pour montrer :

$$\text{si } \texttt{O} = d \text{ alors } \begin{cases} \texttt{True} & \text{si } d = \texttt{O}\\ \texttt{False} & \text{si } d = \texttt{S}\ k,\end{cases}$$

qu'on applique dans le cas où $d = \texttt{S}\ n$, ce qui donnera False, comme demandé.

1. Il faut aussi réaliser un pattern H pour l'extraire de l'hypothèse, ce qui permet à apply de faire correspondre le but et le principe inductif.

4.8 La tactique `inversion`.

La tactique `inversion` applique le principe d'induction sur un but modifié avec des hypothèses d'égalité en plus de l'hypothèse de départ. Par exemple, pour montrer le résultat `even` (`S` (`S` n)) $\to$ `even` n, on introduit, pour obtenir l'hypothèse H : `even` (`S` (`S` n)). On modifie ensuite le but en

$$(\lambda(k : \texttt{even}).\, k = \texttt{S}\ (\texttt{S}\ n) \to \texttt{even}\ n)\ (\texttt{S}\ (\texttt{S}\ n)).$$

sur lequel on applique le principe d'induction `even_ind`. On procède ensuite par réflexivité pour prouver les hypothèses d'égalité générées.

4.9 La « tactique » `Undo`.

Dans notre pieuvre, le mot clé `Undo` permet d'annuler la dernière étape d'une preuve (comme **Rocq**), mais aussi d'annuler n'importe quelle étape (dans le mode preuve ou non), ce n'est donc pas *vraiment* une tactique.

5 BONUS : *ORGANISATION DU PROJET*.

Le projet s'organise en plusieurs dossiers détaillés ci-dessous.

Dossier `base/`. Ce dossier contient la « librairie standard » de pieuvre, avec les définitions de `True`, `False`, `nat`, `list`, `or`, `and`, *etc.* Ces fichiers `*.8pus` sont injectés, au moment de la compilation, dans pieuvre. Ainsi, ils n'ont pas besoin d'être présent au moment de l'exécution, ce qui rend la pieuvre portable.

Dossier `benchmark/`. Ce dossier contient un *benchmark* sur le module `dynamic_dag.ml` du dossier `bib/`.

Dossier `bib/`. Ce dossier contient les structures de données optimisées utilisées dans pieuvre : les arbres rouge-noir, les arbres PATRICIA, la structure de graphe dynamique, *etc.*

Dossier `bin/`. Ce dossier contient le « point de départ » de l'exécutable pieuvre.

Dossier `docs/`. Ce dossier contient la documentation pour pieuvre, c'est-à-dire ce rapport et le diaporama de la soutenance (en PDF et en LaTeX).

Dossier `lib/`. Ce dossier contient le cœur de pieuvre.

- Les fichiers `autocompletion.ml`, `color.ml`, `debug.ml`, `flags.ml` et `print.ml` s'occupent de l'affichage, des *flags* et de l'autocomplétion du REPL.
- Le fichier `execution.ml` s'occupe aussi d'exécuter le code pieuvre donné.
- Les fichiers `environement.ml`, `fresh.ml` et `global.ml` gèrent les variables, les contextes locaux/-globaux, et la gestion du *shadowing*.
- Le fichier `inductive.ml` permet la définition de types inductifs, et le principe inductif associé.
- Les fichiers `infer.ml` et `sorts.ml` permettent l'inférence et la coercition.
- Le fichier `magic.ml` fait de la magie. (Malheureusement, il a été supprimé.)
- Le fichier `parser_expr.ml` convertit une expression parsée en un terme.
- Les fichiers `terms.ml`, `term_manipulation.ml` et `utils.ml` permettent la gestion des termes ;
- Les fichiers `proof.ml`, `tactics_aux.ml` et `tactics.ml` s'occupent de la gestion du mode « preuve » de pieuvre, dont l'exécution des (multiples) tactiques.

Dossier `proofs/`. Ce dossier contient notre banque de tests pieuvre (vérification d'α-équivalence, β-réduction, *type-checking*, et exécution « classique » de fichiers pieuvre).

Dossier `test/`. Ce dossier contient les tests unitaires sur les modules du dossier `bib/`.

La répartition des tâches est la suivante :

- les tactiques et la gestion en mode preuve : Thibaut ;
- l'inférence et l'implémentation du calcul des constructions inductives : Hugo ;
- gestion des noms de variables : Thibaut ;
- l'implémentation (inutilisée) de KMP et BFPRT : Thibaut ;
- la génération du terme du principe inductif : Thibaut ;
- l'extension VSCode « VSPIEUVRE » : Hugo (et ~~Juliette~~ *DeepSeek*).
- ce rapport et la (courte) présentation de soutenance : Hugo.