# Formalisation de Sémantiques Squelettiques

Louis Noizet, Alan Schmitt

# Formalisation de Sémantiques Squelettiques

Louis Noizet and Alan Schmitt

Inria Rennes — Bretagne Atlantique

**Résumé**

Les sémantiques squelettiques sont une approche récente pour décrire et manipuler des sémantiques opérationnelles de langages de programmation. Une description squelettique peut être utilisée pour prouver la correction d'une sémantique abstraite ou pour générer un interpréteur en OCaml. Nous décrivons dans ce travail comment automatiquement extraire d'une description squelettique une formalisation en Coq de sa sémantique naturelle. Cette formalisation peut ensuite être utilisée pour prouver des propriétés sur la sémantique, que nous illustrons de deux manières : par la preuve qu'un programme calcule bien la fonction factorielle et par la preuve de correction d'un compilateur d'expressions arithmétiques vers une machine à pile.

## 1 Introduction

La sémantique squelettique [1] est un métalangage pour spécifier la sémantique d'un langage de programmation. Son principe consiste à ne donner que la *structure* de l'évaluation d'un terme (suite d'opérations, récursion, sélection) sans spécifier l'implémentation concrète des opérations de bases, comme l'addition ou la comparaison à zéro. L'approche squelettique permet de découpler la définition d'un langage de définitions ou preuves de propriétés. Par exemple, il est montré dans [1] comment définir une notion de sémantique abstraite indépendemment du langage, qui est correcte par construction si les opérations de base sont correctes. Plus récemment, un générateur d'un interpréteur en OCaml pour un langage décrit en sémantique squelettique a été présenté [3].

Notre travail décrit une formalisation en Coq de sémantiques squelettiques et de leurs interprétations naturelles [5]. Nous utilisons la sémantique squelettique parce que plusieurs langages ont été ou sont en train d'être formalisés dans ce cadre. Après un rappel du cadre formel des sémantiques squelettiques (Section 2), nous décrivons notre formalisation en Coq de ce cadre (Section 3.1) et de l'interprétation naturelle (Section 3.2). Notre outil, dont le code source est disponible en ligne, [1] traduit des descriptions sémantiques en fichiers Coq utilisables pour faire des preuves formelles sur le langage en utilisant cette formalisation. Nous évaluons notre approche au travers de deux exemples : un compilateur certifié de langages simples et la preuve de correction d'un algorithme avec une boucle `while` (Section 4).

## 2 Sémantique squelettique

Les sémantiques squelettiques ont été développées pour représenter de manière réutilisable et modulaire la structure de sémantiques opérationnelles. L'approche s'appuie sur le fait qu'on peut décomposer la structure d'une sémantique en récursions, branchements et exécutions séquentielles. Le but est ainsi de tirer parti de ces similitudes entre sémantiques pour concevoir un outil modulaire qui puisse servir simultanément à toutes sortes de traitements différents.

$$\frac{\sigma, e \Downarrow \mathit{true} \qquad \sigma, t_1 \Downarrow x_o}{\sigma, \mathit{if}\ e\, t_1\, t_2 \Downarrow x_o} \qquad\qquad \frac{\sigma, e \Downarrow \mathit{false} \qquad \sigma, t_2 \Downarrow x_o}{\sigma, \mathit{if}\ e\, t_1\, t_2 \Downarrow x_o}$$

FIGURE 1 – Règles habituelles pour la sémantique concrète d'un *if*

$$\mathit{eval}(x_\sigma, \mathit{if}\ x_{t_1}\ x_{t_2}\ x_{t_3}) \coloneqq \left[ \mathit{eval}(x_\sigma, x_{t_1})\ ?\triangleright\ x_{f_1}; \begin{pmatrix} \texttt{isTrue}\,(x_{f_1})\,; \mathit{eval}(x_\sigma, x_{t_2})\ ?\triangleright\ x_o \\ \texttt{isFalse}\,(x_{f_1})\,; \mathit{eval}(x_\sigma, x_{t_3})\ ?\triangleright\ x_o \end{pmatrix}_{\{x_o\}} \right]_{x_o}$$

FIGURE 2 – Squelette pour la structure *if*

## 2.1 Exemple

La figure 1 décrit l'interprétation usuelle en sémantique naturelle, ou à grands pas, d'une structure de contrôle conditionnelle. En sémantique squelettique, ce même comportement est décrit dans la figure 2.

Décomposons ce « squelette ».

- $\mathit{eval}(x_\sigma, \mathit{if}\ x_{t_1}\ x_{t_2}\ x_{t_3}) \coloneqq_{\ldots}$ : on définit l'évaluation de l'expression *if* $x_{t_1}\ x_{t_2}\ x_{t_3}$ dans l'état $x_\sigma$ par la fonction d'évaluation *eval*. Cette évaluation est une liste `[ ... ]` d'actions séparées par des `;`.
- $\mathit{eval}(x_\sigma, x_{t_1})\ ?\triangleright\ x_{f_1}$ : la première action appelle récursivement l'évaluation de $x_{t_1}$ dans l'état $x_\sigma$, et on affecte le résultat à $x_{f_1}$. On utilise la fonction d'évaluation *eval*, que l'on appelle un *crochet*.
- $\begin{pmatrix} \texttt{isTrue}\,(x_{f_1})\,; \mathit{eval}(x_\sigma, x_{t_2})\ ?\triangleright\ x_o \\ \texttt{isFalse}\,(x_{f_1})\,; \mathit{eval}(x_\sigma, x_{t_3})\ ?\triangleright\ x_o \end{pmatrix}_{\{x_o\}}$ : cette deuxième action est un *branchement*. On évalue chaque ligne séparément, puis on affecte à la variable $x_o$ le résultat de l'évaluation (la manière de calculer le résultat global à partir du résultat de chaque branche dépend de l'interprétation choisie, comme détaillé en Section 2.3). Le $\{x_o\}$ en indice du branchement désigne l'ensemble des valeurs qui sont définies et donc utilisables à l'issue du branchement.
- `isTrue`$(x_{f_1})$ : on calcule si $x_{f_1}$ est vrai. Les primitives du type `isTrue` et `isFalse` sont appelées des *filtres*. Leur comportement n'est pas spécifié dans la sémantique. Chaque filtre prend en argument et renvoie en résultat un certain nombre (fini) de valeurs, possiblement aucune, comme c'est le cas ici. Dans ce cas, le filtre agit comme un sélecteur, les branches étant conservées ou jetées en fonction des arguments donnés au filtre.
- $\mathit{eval}(x_\sigma, x_{t_2})\ ?\triangleright\ x_o$ : on évalue récursivement le terme $x_{t_2}$.
- $[]_{x_o}$ : le $x_o$ en indice à la fin du squelette est la valeur qui est renvoyée par le squelette.

On peut observer que cette représentation est très calculatoire : à l'exception des branches, dont on ne donne pas l'ordre d'évaluation, on décrit intuitivement un évaluateur pour la sémantique.

On peut ensuite donner plusieurs *interprétations* possibles au squelette en fonction de la

1. https://gitlab.inria.fr/skeletons/necro/tree/JFLA_2020

sémantique que l'on veut obtenir. Par exemple, l'interprétation concrète fournit une sémantique naturelle, et l'interprétation abstraite peut être utilisée pour prouver que des analyses sont correctes. On peut aussi utiliser une sémantique squelettique pour générer un interpréteur (comme le logiciel necro [2]), un analyseur statique, etc.

Les sémantiques squelettiques sont volontairement simples. Par exemple, elles ne fournissent pas de manière primitive de représenter des lieurs. Si on veut modéliser un langage avec lieur, il faut déclarer, typiquement avec des filtres, des opérations de substitution ou des opérations manipulant des environnements. [3] Cette simplicité offre une grande flexibilité dans la manière dont on définit les sémantiques. Nous ne montrons dans ce travail que des sémantiques à grands pas, mais il est aussi possible de définir des sémantiques à petits pas.

## 2.2 Définition formelle

Une sémantique squelettique se compose de plusieurs déclarations : les termes, les sortes et les règles.

Les sortes des termes se décomposent en *sortes de base* représentant les éléments de base de notre langage (par exemple les littéraux) et les *sortes de programme* qui correspondent aux termes construits, décrits ci-après. Les termes construits sont ceux qui ont un constructeur en tête et qui seront évalués. On déclare également des *sortes de flux* pour les objets qui seront présents à l'interprétation d'un programme, comme des valeurs, un état ou une pile.

Un terme est une variable $x_i$ ou un constructeur appliqué à des termes. Les termes de bases ne sont instantiés que lorsqu'une interprétation est donnée.

$$\textsc{Terme} \quad t ::= x_i \mid c(t..t)$$

Un squelette a la forme $S_{[x_1..x_n]}$ où $S$ est un corps de squelette

$$\begin{aligned} \textsc{Corps} \quad & S ::= [] \mid B; S \\ \textsc{Os} \quad & B ::= h\,(x_1..x_n, t)\ ?\triangleright\ (y_1..y_m) \mid f\,(x_1..x_n)\ ?\triangleright\ (y_1..y_m) \mid (S..S)_V \end{aligned}$$

Un corps de squelette est une succession d'os. Chaque os est un crochet, un filtre ou un branchement. Un crochet $h(x_1..x_n, t)\ ?\triangleright\ (y_1..y_m)$ correspond à l'évaluation récursive de la sémantique du terme $t$ dans l'état $(x_1..x_n)$, et à l'affectation de son résultat aux variables $(y_1..y_m)$. Un filtre $f(x_1..x_n)\ ?\triangleright\ (y_1..y_n)$ examine si les $x_i$ vérifient certaines pré-conditions et le cas échéant, renvoie des valeurs à affecter aux $y_i$. Un branchement est du type $(S_1..S_n)_V$ où $V$ est un ensemble de variables qui sont définies dans chaque branche et utilisables à la suite de l'exécution du branchement.

Pour tout ensemble $E$, on note $E^*$ l'ensemble des suites finies d'éléments de $E$. Formellement, $E^* := \cup_{n\in\mathbb{N}} E^n$. Donner la sémantique squelettique d'un langage correspond à donner :

- l'ensemble $\Sigma_f$ des sortes de flux, l'ensemble $\Sigma_b$ des sortes de base et l'ensemble $\Sigma_p$ des sortes de programme (on pose $\Sigma = \Sigma_f \cup \Sigma_b \cup \Sigma_p$, et on distingue les sortes de termes $\Sigma_t = \Sigma_b \cup \Sigma_p$) ;
- l'ensemble $C$ des constructeurs, et une fonction csort : $C \to \Sigma_t^* \times \Sigma_p$ qui donne les sortes attendues et la sorte de retour (on note $\text{csort}_1$ et $\text{csort}_2$ ses deux projections) ;
- un ensemble $F$ de filtres, et une fonction fsort : $F \to \Sigma^* \times \Sigma^*$, qui donne les sortes attendues et les sortes de retour ;

2. https://gitlab.inria.fr/skeletons/necro/
3. https://gitlab.inria.fr/skeletons/necro/blob/JFLA_2020/test/lambda_rules.txt

— un ensemble $H$ de crochets, et une fonction hsort : $H \to \Sigma_f^* \times \Sigma_p \times \Sigma_f^*$, qui donne les sortes attendues et les sortes de retour (on note $\text{hsort}_1$, $\text{hsort}_2$ et $\text{hsort}_3$ ses projections) ;
— pour chaque crochet $h \in H$, un ensemble de règles de la forme : $h(y_1..y_n, c(x_{t_1}..x_{t_m})) \coloneqq S_{(z_1..z_p)}$ où $c(x_{t_1}..x_{t_m})$ est le terme évalué ($c \in C$ et $\text{csort}_2(c) = \text{hsort}_2(h)$), les $y_1..y_n$ donnent l'état dans lequel on évalue le terme $c(x_{t_1}..x_{t_m})$ et $S_{(z_1..z_p)}$ est un squelette.

La syntaxe telle qu'elle est décrite ci-dessus ne correspond pas tout à fait à celle définie dans l'article [1], car la sémantique squelettique étant une notion très récente, sa syntaxe est encore en cours d'évolution. Par exemple, on manipule maintenant plusieurs crochets alors que l'article initial n'en définissait qu'un. Ainsi, on peut évaluer une expression de plusieurs manières différentes suivant son contexte. Par ailleurs, la syntaxe initiale ne permettait d'avoir qu'une seule variable en entrée et en sortie d'un crochet, alors que l'on s'autorise maintenant à en avoir un nombre arbitraire. Cela évite d'avoir à systématiquement construire et détruire des n-uplets.

**Bonne formation** La présentation originelle des sémantiques squelettiques proposait une interprétation de *bonne formation*, qui vérifie que les crochets et les filtres utilisent leurs arguments et valeurs de retour de manière cohérente en ce qui concerne leurs sortes. Pour ce faire, la bonne formation doit se rappeler des sortes des variables squelettiques précédemment définies. Notre formalisation utilise une approche légèrement différente, où nous annotons chaque variable squelettique par sa sorte, comme détaillé en Section 3.1.

## 2.3 Interprétation concrète

On définit ensuite l'interprétation concrète, correspondant à la sémantique naturelle [5], de cette sémantique. On définit donc la sémantique d'un terme avec des arbres de dérivation qui représentent les appels récursifs. L'idée est donc de commencer par les squelettes sans récursion (sans crochet), qui sont les plus simples. Puis on étend au fur et à mesure en ajoutant les interprétations dont les dérivations sont de plus en plus grandes. Pour ce faire, on définit l'interprétation concrète comme le plus petit point fixe d'une fonctionnelle $\mathcal{H}$ qui à un ensemble d'interprétations déjà prouvées (sous forme de quadruplet (crochet, état, terme, résultat)) associe les dérivations qui peuvent utiliser ces interprétations comme prémisses.

Formellement, un terme clos est un terme dans lequel n'intervient aucune variable. Pour chaque sorte de base $b$ correspond un ensemble de termes concrets de base, qui peuvent être utilisée pour construire des termes concrets clos. À chaque sorte de flux $s$ correspond un ensemble concret de valeurs de flux $V_s$. On notera $V_{(s_1..s_n)} \coloneqq V_{s_1} \times \cdots \times V_{s_n}$. Une valeur est ou bien une valeur de flux, c'est-à-dire un élément d'un $V_s$, ou bien un terme clos. Par extension, on notera $V_t$ l'ensemble des termes clos de sorte $t$.

Pour chaque filtre $f$ tel que $\text{fsort}(f) = (s, t)$ où $s$ et $t$ sont des listes de sortes, on définit son interprétation $[\![f]\!] \in V_s \times V_t$. On note $[\![f]\!]\,(v) \Downarrow w$ pour dire que $(v, w) \in [\![f]\!]$. Pour interpréter un squelette, on va exécuter séquentiellement ses os en mettant à jour un environnement $E$ qui contient les valeurs déjà calculées — $E$ est donc une fonction partielle des variables squelettiques vers des valeurs — et on a également besoin d'un ensemble $Q$ qui contient l'ensemble des résultats des dérivations déjà prouvées, sous la forme de quadruplets concrets $(h, v, t, w)$ où $h \in H$, $t$ est un terme clos et $v, w$ sont des listes de valeurs.

À chaque os $B$ correspond une relation $[\![B]\!]$ qui associe la paire d'un environnement et d'un ensemble $Q$ de quadruplets concrets à son environnement mis à jour et au même ensemble $Q$. À chaque squelette $S$ correspond une relation $[\![S]\!]$ qui relie une paire du même type que ci-dessus à une valeur de sortie. Ces interprétations sont définies telles que :

$$[\![f\,(x_1..x_m)\,?\triangleright\,(y_1..y_n)]\!]\,(E, Q) \Downarrow (E' = E + y_1 \mapsto v_1..y_n \mapsto v_n, Q)$$

lorsque $[\![f]\!]\,(E(x_1)..E(x_m)) \Downarrow (v_1..v_n)$ ;

$$[\![h\,(x_1..x_m, t)\ ?\triangleright\ (y_1..y_n)]\!]\,(E, Q) \Downarrow (E' = E + y_1 \mapsto v_1..y_n \mapsto v_n, Q)$$

lorsque $(h, (E(x_1)..E(x_m)), E(t), (v_1..v_n)) \in Q$ ;

$$\left[\!\!\left[(S_1..S_m)_{\{x_1..x_n\}}\right]\!\!\right]\,(E, Q) \Downarrow (E' = E + x_1 \mapsto v_1..x_n \mapsto v_n, Q)$$

lorsqu'il existe $i$ tel que $\left[\!\!\left[[S_i]_{(x_1..x_m)}\right]\!\!\right]\,(E, Q) \Downarrow (v_1..v_n)$ ;

$$\left[\!\!\left[[]_{(x_1..x_n)}\right]\!\!\right]\,(E, Q) \Downarrow (v_1..v_n)$$

lorsque $\forall i, E(x_i) = v_i$ ;

$$\left[\!\!\left[[B_1; ..; B_n]_{(x_1..x_m)}\right]\!\!\right]\,(E, Q) \Downarrow O$$

lorsque $[\![B_1]\!]\,(E, Q) \Downarrow (E', Q')$ et $\left[\!\!\left[[B_2; ..; B_n]_{(x_1..x_m)}\right]\!\!\right]\,(E', Q') \Downarrow O$.

Alors, on définit la fonctionnelle de conséquence immédiate $\mathcal{H}$ de la manière suivante :

$$\mathcal{H}\,(Q) = \left\{ (h, (\sigma_1..\sigma_n)\,, t, (v_1..v_m)) \;\middle|\; \begin{array}{c} t = c\,(t_1..t_p) \wedge \mathrm{csort}_2(c) = \mathrm{hsort}_2(h) \\ (t_1..t_p) : \mathrm{csort}_1(c) \\ h\big(x_{\sigma_1}..x_{\sigma_n}, c\,\big(x_{t_1}..x_{t_p}\big)\big) \coloneqq S_{[y_1..y_m]} \\ (\sigma_1..\sigma_n) : \mathrm{hsort}_1(h) \\ \Sigma = x_{\sigma_1} \mapsto \sigma_1..x_{\sigma_n} \mapsto \sigma_n + x_{t_1} \mapsto t_1..x_{t_p} \mapsto t_p \\ [\![S]\!]\,(\Sigma, Q) \Downarrow \Sigma' \\ \forall i = 1..m,\ \Sigma'\,(y_i) = v_i \end{array} \right\}.$$

Cette fonctionnelle construit un ensemble de quadruplets $(h, (\sigma_1..\sigma_n), t, (v_1..v_m))$ de la manière suivante. Elle extrait le constructeur de tête du terme $t$, elle vérifie qu'il est compatible avec le crochet considéré et que les sous-termes sont compatibles avec le constructeur, elle trouve la règle associée au crochet et au constructeur, elle vérifie que les valeurs d'entrées $(\sigma_1..\sigma_n)$ sont compatibles avec le crochet, elle évalue le squelette dans un environnement initial et enfin elle valide que les valeurs retournées sont $(v_1..v_m)$.

Ainsi, $\mathcal{H}^n(\emptyset)$ contient l'ensemble des évaluations qui peuvent être obtenues par des dérivations de taille au plus $n$. L'interprétation concrète $\Downarrow$ est alors définie par $\Downarrow\ = \bigcup_n \mathcal{H}^n(\emptyset)$.

Prenons l'exemple du *if* (figure 2). On veut calculer la sémantique de $\mathit{eval}(\sigma, \mathit{if}\ t_1\, t_2\, t_3)$. On pose $E = x_\sigma \mapsto \sigma + x_{t_1} \mapsto t_1..x_{t_3} \mapsto t_3$, et on suppose que $Q$ contient déjà $(\mathit{eval}, \sigma, t_1, \mathit{false})$ et $(\mathit{eval}, \sigma, t_3, v)$. Alors on a :

$$[\![\mathit{eval}\,(x_\sigma, x_{t_1})\ ?\triangleright\ x_{f_1}]\!]\,(E, Q) \Downarrow (E' = E + x_{f_1} \mapsto \mathit{false}, Q).$$

On entre dans une branche donc on fait séparément les deux branches. `isTrue` va jeter `false` et `isFalse` va le conserver : On a

$$[\![\texttt{isFalse}\,(\texttt{false})]\!]\,(E', Q) \Downarrow (E', Q).$$

Puis, on a

$$[\![\mathit{eval}\,(x_\sigma, x_{t_3})\ ?\triangleright\ x_o]\!]\,(E', Q) \Downarrow (E'' = E' + x_o \mapsto v, Q).$$

Alors, on en déduit que

$$\left[\!\!\left[ \begin{pmatrix} \texttt{isTrue}\,(x_{f_1})\,; eval\,(x_\sigma, x_{t_2})\ ?\triangleright\ x_o \\ \texttt{isFalse}\,(x_{f_1})\,; eval\,(x_\sigma, x_{t_3})\ ?\triangleright\ x_o \end{pmatrix}_{\{x_o\}} \right]\!\!\right] (E', Q) \Downarrow (E'' = E' + x_o \mapsto v, Q).$$

Ainsi, on peut conclure que $(eval, \sigma, \mathit{if}\ t_1\, t_2\, t_3, v) \in \mathcal{H}(Q)$.

# 3 Formalisation en Coq

La formalisation en Coq d'une sémantique squelettique comporte trois familles de fichiers. Tout d'abord, les fichiers écrits une fois pour toute et décrivant le cadre formel : la définition des squelettes (Section 3.1) et la définition d'une interprétation concrète (Section 3.2). Ensuite, des fichiers générés automatiquement à partir de la description squelettique de la sémantique. Ces fichiers instancient la définition générale des squelettes pour la sémantique considérée. Enfin, des fichiers supplémentaires fournissent l'implémentation des filtres, ou donnent des propriétés sur ceux-ci, en fonction de l'application considérée. Cette troisième famille de fichiers sera décrite en Section 4.

## 3.1 Squelettes

Ce premier fichier Skeleton.v formalise la syntaxe des sémantiques squelettiques ainsi que la notion de bonne formation.

On suppose donnés tous les éléments de base d'une sémantique squelettique (l'ensemble des sortes $\Sigma$, l'ensemble des constructeurs, filtres, crochets et les fonctions csort, fsort et hsort). Alors on donne la définition d'un terme de sorte `s` (on exclut volontairement les termes de base, qui n'existent pas au niveau de la sémantique squelettique, mais qui seront instanciés au niveau des programmes et des interprétations).

```
Inductive skel_var : sort -> Type :=
| skel_var_intro : forall s, string -> skel_var s.

Inductive term : term_sort -> Type :=
| term_sv : forall t, skel_var (Term t) -> term t
| term_constructor : forall (c:constructor),
    list_term (fst (csort c)) -> term (Prgm (snd (csort c)))
with list_term: list term_sort -> Type :=
| nil_term: list_term []
| cons_term: forall a A, term a -> list_term A -> list_term (a::A).
```

Les définitions d'un os et d'un squelette sont exactement les mêmes que dans la formalisation ci-dessus, à l'exception des variables squelettiques qui sont explicitement typées (leur sorte est donnée) :

```
Inductive bone :=
| H : hook -> list typed_skel_var -> typed_term -> list typed_skel_var -> bone
| F : filter -> list typed_skel_var -> list typed_skel_var -> bone
| B : list (list bone) -> list typed_skel_var -> bone.
Definition skeleton := list (bone).
```

Une sémantique squelettique consiste enfin en la donnée d'une liste de règles du type $h(y_1..y_n, c(x_{t_1}..x_{t_n})) \coloneqq S_{(z_1..z_p)}$, modélisées par le uplet $(h, [y_1..y_m], c, [x_{t_1}..x_{t_p}], S, [z_1..z_p])$, où les variables squelettiques sont elles aussi typées.

```
Definition skeletal_semantics :=
  list (hook * list typed_skel_var * constructor * list typed_skel_var *
        skeleton * list typed_skel_var).
```

Par ailleurs, en utilisant le typage des termes, on définit une fonction `well_formed` à valeurs booléennes, qui vérifie qu'une sémantique est bien formée (code). Enfin, on définit un langage comme un enregistrement contenant tous les ensembles et fonctions précédemment cités, la liste représentant la sémantique squelettique, ainsi que la preuve que la sémantique est bien formée (si on donne une sémantique bien formée, le résultat est immédiat par réflexivité). Le générateur vérifie déjà la bonne formation, mais celui-ci étant écrit en OCaml, la vérification en Coq donne des garanties supplémentaires.

```
Record language := mklang
  {  l_cons: Type;
     l_filter: Type;
     l_hook: Type;
     l_base_sort: Type;
     l_flow_sort: Type;
     l_prgm_sort: Type;
     l_base_sort_eq_dec: forall x y: l_base_sort, {x = y} + {x <> y};
     l_flow_sort_eq_dec: forall x y: l_flow_sort, {x = y} + {x <> y};
     l_prgm_sort_eq_dec: forall x y: l_prgm_sort, {x = y} + {x <> y};
     l_csort: l_cons ->
        ( list (term_sort l_base_sort l_prgm_sort) *
          l_prgm_sort);
     l_fsort: l_filter ->
        ( list (sort l_base_sort l_flow_sort l_prgm_sort) *
          list (sort l_base_sort l_flow_sort l_prgm_sort));
     l_hsort: l_hook ->
        ( list l_flow_sort * l_prgm_sort * list l_flow_sort );
     l_semantics: skeletal_semantics l_cons l_filter l_hook l_base_sort
                    l_flow_sort l_prgm_sort l_csort;
     l_semantics_well_formed: well_formed l_cons l_filter l_hook l_base_sort
                                l_flow_sort l_prgm_sort l_base_sort_eq_dec
                                l_flow_sort_eq_dec l_prgm_sort_eq_dec l_csort
                                l_fsort l_hsort l_semantics
                              = true}.
```

Ce type enregistrement est instancié par les fichiers générés automatiquement, qui se contentent de créer un enregistrement de type `language` (voir par exemple While.v). Par conséquent, le typage automatique des variables squelettiques ne requiert pas d'effort supplémentaire, car ces types sont inférés par l'outil de génération.

## 3.2 Interprétation Concrète

Nous formalisons maintenant l'interprétation concrète d'un squelette. On suppose donnés un langage, et des fonctions qui définissent l'interprétation des sortes de bases et des sortes de

flux :

```
  Variable l: language.
  Variable base_interp: l.(l_base_sort) -> Type.
  Variable flow_interp: l.(l_flow_sort) -> Type.
```

Les termes concrets sont des termes clos, sans variables de termes, qui représentent des termes du langage. Un terme concrete est soit un terme de base, soit un constructeur appliqué à des termes.

```
Inductive cterm :=
| cterm_base : forall (b:l.(l_base_sort)), base_interp b -> cterm
| cterm_constructor : l.(l_cons) -> list cterm -> cterm.
```

On définit aussi les valeurs, et les quadruplets concrets.

```
Inductive value :=
| val_flow : forall f, flow_interp f -> value
| val_cterm : cterm -> value.

Definition concrete_quadruple := (l.(l_hook) * list value * cterm *
                                  list value)%type.
```

On suppose maintenant qu'on a une interprétation des filtres (donc une fonction qui à chaque filtre associe une relation).

```
Variable filter_interp  : l.(l_filter) -> list value -> list value -> Prop.
```

Enfin, on définit l'interprétation des squelettes, des os, et la sémantique concrète. Deux versions existent, dans deux fichiers différents. La première (Concrete.v) utilise l'induction structurelle de Coq pour interpréter les crochets. La seconde (Concrete2.v) suit la définition donnée en Section 2.3 et définit une fonctionnelle de *conséquence immédiate*, l'homologue de la fonction $\mathcal{H}$ définie plus haut, qui transforme un ensemble de quadruplets en un autre ensemble de quadruplets. La sémantique en est alors le plus petit point fixe.

Voici la définition principale de la première version (code).

```
Inductive interp_skel : skeleton -> env -> env -> Prop :=
| i_Cons : forall B S s1 s2 s3,
    interp_bone B s1 s2 ->
    interp_skel S s2 s3 ->
    interp_skel (B::S) s1 s3
| i_Void : forall s, interp_skel [] s s

with interp_bone : bone -> env -> env -> Prop :=
| i_F : forall lv f (f1 : env) (l1 : list typed_skel_var) l2,
    filter_interp_opt f (List.map (find f1) (map skel_name l1)) lv ->
    interp_bone (F f l1 l2) f1 (add_asn f1 (l2) lv)
| i_H : forall (v1 v2: list value) (h: l.(l_hook)) (ct: cterm)
               (t : typed_term) (e:env) (xf1 xf2:list typed_skel_var),
    eval_term e (projT2 t) = Some ct ->
    map (find e) (map skel_name xf1) = map Some v1 ->
```

```
    concrete_semantics (h,v1,ct,v2) ->
    interp_bone (H h xf1 t xf2) e (add_asn e xf2 v2)
| i_B : forall vals Ss Si V e e',
    List.In Si Ss -> interp_skel Si e e' ->
    map (find e') (map (skel_name) V) = map Some vals ->
    interp_bone (B Ss V) e (add_asn e V vals)

with concrete_semantics :concrete_quadruple -> Prop :=
| cs_intro:
    forall (h: l.(l_hook)),
    forall (c: l.(l_cons)) (s xt xo: list typed_skel_var),
    forall (S: skeleton),
    forall lt s1 s2 sigma,
    In (h,s,c,xt,S,xo) (l.(l_semantics)) ->
    interp_skel S (add_asn (add_asn_cterm void_env xt lt) s s1) sigma ->
    unfold_list_option (map (find sigma) (map skel_name xo)) = Some s2 ->
    concrete_semantics (h, s1, cterm_constructor c lt, s2).
```

Le cas intéressant de cette définition est pour le crochet, qui utilise le prédicat `concrete_semantics`, défini en induction mutuelle, pour la récursion.

Voici la définition principale de la seconde version (code).

```
Inductive interp_skel : skeleton -> Cstate -> Cresult -> Prop :=
| i_Cons : forall B S s1 s2 T r,
    interp_bone B (s1,T) s2 ->
    interp_skel S (s2,T) r ->
    interp_skel (B::S) (s1,T) r
| i_Void : forall s t, interp_skel [] (s,t) s

with interp_bone : bone -> Cstate -> Cresult -> Prop :=
| i_F : forall f (f1 : env) tl lv (l1 : list typed_skel_var) l2,
    filter_interp_opt f (List.map (find f1) (map skel_name l1)) lv ->
    interp_bone (F f l1 l2) (f1,tl) (add_asn f1 (l2) lv)
| i_H : forall (h: (l.(l_hook))) (e : env) (T:concrete_quadruple -> Prop)
    (xf1 xf2 : list typed_skel_var) (t:typed_term) (v : list value),
    match unfold_list_option (map (find e) (map skel_name xf1)),
          eval_term e (projT2 t) with
    | Some xf1', Some t' => T (h, xf1', t', v)
    | _, _ => False
    end -> interp_bone (H h xf1 t xf2) (e,T) (add_asn e xf2 v: env)
| i_B : forall Ss Si V e T vals e',
    List.In Si Ss -> interp_skel Si (e,T) e' ->
    map (find e') (map skel_name V) = map Some vals ->
    interp_bone (B Ss V) (e,T) (add_asn e V vals).

Inductive immediate_consequence : (concrete_quadruple -> Prop) ->
                                  concrete_quadruple -> Prop :=
| H_intro :
    forall (h: l.(l_hook)),
```

```
    forall (c: l.(l_cons)) (s xt xo: list typed_skel_var),
    forall (S: skeleton),
    forall lt s1 s2 T sigma,
    In (h,s,c,xt,S,xo) l.(l_semantics) ->
    interp_skel S (add_asn (add_asn_cterm void_env xt lt) s s1, T) sigma ->
    unfold_list_option (map (find sigma) (map skel_name xo)) = Some s2 ->
    immediate_consequence T (h, s1, cterm_constructor c lt, s2).

Fixpoint consequence n :=
  match n with
  | 0 => fun _ => False
  | S m => immediate_consequence (consequence m)
  end.

Definition concrete_semantics (t : concrete_quadruple) : Prop :=
  exists n, consequence n t.
```

Dans cette version, le crochet s'appuie sur l'ensemble de quadruplets passé en argument pour définir la récursion.

L'équivalence entre ces deux versions est immédiate, et est également prouvée (code). Cependant, on a choisi de garder ces deux versions qui offrent chacune des facilités suivant l'usage prévu. En particulier, pour les deux applications proposées ci-dessous, on utilise tantôt la première version (Section 4.3) et tantôt la seconde (Section 4.2).

### 3.3 Générateur

Le générateur transforme un fichier de syntaxe type necro en un fichier coq qui produit un enregistrement de type `language`.

On appelle donc un analyseur lexical et un analyseur syntaxique de fichiers de sémantique squelettique, puis un typeur qui vérifie la bonne formation des crochets et ajoute des annotations de type. Ces trois outils sont ceux qui ont été définis pour necro, mentionné ci-dessus.

Ensuite, un fichier prérempli est défini (code), et on se contente de générer les parties manquantes avec un ensemble de fonctions de génération (code).

## 4 Applications

### 4.1 Exemple de Génération

Prenons l'exemple de l'opérateur conditionnel `if` défini ci-dessus. Sa version définie en sémantique squelettique s'écrit de la manière suivante (code).

```
If (x_t1, x_t2, x_t3) ->
  x_f1  <- expr x_s x_t1;   (* Appel de crochet *)
  x_f1' <- isBool x_f1;     (* Appel de filtre *)
  x_o <- branch             (* Branchement *)
    isTrue x_f1';           (* Appel de filtre *)
    x_o <- stmt x_s x_t2    (* Appel de crochet *)
  or                        (* Deuxième branche *)
    isFalse x_f1';          (* Appel de filtre *)
```

```
  x_o <- stmt x_s x_t3    (* Appel de crochet *)
end;                      (* Fin du branchement *)
x_o                       (* Valeur renvoyée *)
```

Cette définition est traduite automatiquement par notre outil en ce quadruplet (code).

```
(h_stmt, [tsvi (Flow s_state) "x_s"],
 c_If,
 [tsvi (Term (Prgm s_expr)) "x_t1"; tsvi (Term (Prgm s_stmt)) "x_t2";
  tsvi (Term (Prgm s_stmt)) "x_t3"],
 [H h_expr [tsvi (Flow s_state) "x_s"]
           (tterm_sv (svi (Term (Prgm s_expr)) "x_t1"))
           [tsvi (Flow s_value) "x_f1"] ;
    F f_isBool [tsvi (Flow s_value) "x_f1"] [tsvi (Flow s_vbool) "x_f1'"];
    B [[F f_isTrue [tsvi (Flow s_vbool) "x_f1'"] [];
        H h_stmt [tsvi (Flow s_state) "x_s"]
          (tterm_sv (svi (Term (Prgm s_stmt)) "x_t2"))
          [tsvi (Flow s_state) "x_o"]];
       [F f_isFalse [tsvi (Flow s_vbool) "x_f1'"] [];
        H h_stmt [tsvi (Flow s_state) "x_s"]
          (tterm_sv (svi (Term (Prgm s_stmt)) "x_t3"))
          [tsvi (Flow s_state) "x_o"]]]
      [tsvi (Flow s_state) "x_o"]],
    [tsvi (Flow s_state) "x_o"])
```

Il est intéressant de remarquer que les `tsvi` sont les variables squelettiques, annotées par leurs sortes inférées par l'outil.

## 4.2 Compilation certifiée

Prenons un langage arithmétique et un langage à pile :

```
expr ::=                        stmt ::=
| Const <literal>               | Skip
| Plus <expr> <expr>            | Seq <stmt> <stmt>
| Minus <expr> <expr>           | Push <literal>
| Times <expr> <expr>           | Add | Sub
| Over <expr> <expr>            | Mul | Div
```

On définit leur sémantique squelettique dans les fichiers Arithmetic.sk et Stack.sk. La compilation du premier langage vers le deuxième est définie dans Compile.v.

Le compilateur est prouvé sans choisir d'implémentation concrète pour les sortes et les filtres. Seules sont exigées certaines propriétés sur ceux-ci, que l'on définit comme des axiomes dans le fichier Prelim.v. Par exemple, on demande que les sortes `value` des deux langages soient interprétées de manière similaire, donc qu'il y ait un isomorphisme entre les deux interprétations (code). Par ailleurs, on exige que l'interprétation du filtre `push` soit l'inverse à gauche de celle du filtre `pop` (code). On demande également que les opérations arithmétiques soient interprétées de manière similaire (code) dans les deux langages.

Une fois posés ces axiomes, on peut prouver le résultat suivant (code) qui affirme que les sémantique d'un programme et de sa version compilée sont équivalentes.

```
forall i1 o1 i2 o2, StateRel i1 o1 i2 o2 ->
forall e e', compile e = Some e' ->
  (exists h, concrete_semantics ar bia fia Afilter_interp (h, i1, e, o1)) <->
  exists h, concrete_semantics st bis fis Sfilter_interp (h, i2, e', o2).
```

Pour prouver ce résultat, on choisit le deuxième fichier d'interprétation concrète, qui définit l'induction comme le plus petit point fixe de la fonctionnelle de conséquence immédiate, pour pouvoir prouver le théorème de correction par récurrence sur la hauteur de l'arbre de dérivation (code). Le résultat est assez immédiat.

## 4.3 Certification de programme

On prend un langage while dont les expressions et déclarations sont définies de la manière suivante.

```
expr ::=                        stmt ::=
| Const <literal>               | Skip
| Var <ident>                   | Assign <ident> <expr>
| Plus <expr> <expr>            | Seq <stmt> <stmt>
| Equal <expr> <expr>           | If <expr> <stmt> <stmt>
| Not <expr>                    | While <expr> <stmt>
```

La sémantique squelettique de ce langage est définie dans le fichier While.sk. On choisit une implémentation concrète du langage. Pour cela, il faut d'abord choisir les termes de base. On prend les chaînes de caractères comme identifiants, et les entiers relatifs comme littéraux. De même, on choisit une implémentation pour les sortes de flot (code). Enfin, on définit également l'interprétation concrète des filtres (code). On écrit dans ce langage un programme pour calculer la factorielle de $n$ (pour alléger les notations, on notera `a;b` pour `Seq a b`).

```
Assign "over" (Const 0);
Assign "fact" (Const 1);
Assign "n" (Const n);
Assign "a" (Const 0);
Assign "i" (Const 0);
While (Not (Equal (Var "n") (Const 0))) (
  Assign "a" (Var "fact");
  Assign "i" (Const 1);
  While (Not (Equal (Var i) (Var n))) (
    Assign "fact" (Plus (Var "fact") (Var "a"));
    Assign "i" (Plus (Var "i") (Const 1))
  );
  Assign "n" (Plus (Var "n") (Const -1))
)
```

La démonstration est ensuite une démonstration classique pour ce genre de résultat. On commence par définir deux lemmes correspondant aux invariants des deux boucles et on montre le résultat suivant par récurrence (code), où `rd_fm_state` accède à une variable de l'état, et où `fact` calcule (mathématiquement) la factorielle.

```
  exists (x:flow_interp s_state), while_semantics
  ( h_stmt,
```

```
    [val_flow s_state init_state],
    fact_prog n,
    [val_flow s_state x] )
  /\ rd_fm_state x v_fact = Some (Int (fact n)).
```

Puis on applique ces deux lemmes pour conclure (code). Cette fois, on choisit l'interprétation concrète utilisant l'induction de Coq, car elle permet plus facilement d'automatiser les calculs. On note par ailleurs que l'initialisation des variables `"a"` et `"i"` a pour unique but de simplifier l'énoncé du lemme de la boucle principale.

# 5 Travaux Connexes

Les travaux les plus proches des nôtres sont bien entendu la formalisation en Coq de [1]. Les raisons pour lesquelles nous n'avons pas réutilisé ce travail sont les suivantes. Tout d'abord, cette formalisation est très générique : la notion même d'interprétation est formalisée, indépendemment des interprétations concrètes et abstraites. Cela requiert l'introduction de notions supplémentaires qui alourdissent la définition de l'interprétation concrète et qui rendent plus complexe le raisonnement sur celle-ci. De plus, dans un souci de lisibilité du code généré, nous avons préféré nous éloigner de la définition formelle initiale de l'article pour simplifier les définitions. Nous avons montré en section 3.2 l'équivalence de deux interprétations concrètes. Nous pouvons étendre ce résultat pour montrer l'équivalence entre notre formalisation et celle de [1].

Ott [9] est un système permettant de décrire la sémantique de langages de programmation et leurs systèmes de types. Ott utilise une syntaxe accessible définissant ces sémantiques sous forme de règles d'inférences. Il fournit des outils pour traduire ces définitions en interpréteurs exécutables et en spécification dans les assistants de preuve Coq et HOL. Ott propose plus de fonctionnalités que notre approche, comme la gestion des lieurs, mais il est moins flexible dans la génération de code. En particulier, les traits inductifs d'une sémantique sont nécessairement représentés par un inductif en Coq, et la taille de cet inductif est proportionnelle à la taille de la sémantique. L'utilisation d'autres représentations peut être utile, comme nous l'avons vu en Section 4.2, mais également pour faire des preuves sur de grosses sémantiques.

Le système $\mathbb{K}$ [8] définit un cadre formel pour décrire des sémantiques opérationnelles et pour construire des vérificateurs de programmes s'appliquant directement aux définitions sémantiques sans passer par une représentation intermédiaire ou une logique de programme. Les règles sémantiques sont écrites sous la forme de règles de réécriture s'appliquant à un état sémantique. Le système $\mathbb{K}$ a été utilisé pour décrire les sémantiques de plusieurs langages, comme C, Java, ou JavaScript. Le vérificateur de programme s'appuie sur la « matching logic » [7], un système formel permettant de raisonner sur les motifs et les ensembles de termes associés. Cette approche est efficace mais particulièrement complexe : la sémantique de l'outil lui-même est peu documentée [6] et la génération de code pour d'autres outils de preuve n'existe que sous forme de prototype.

Plusieurs outils se focalisent sur la vérification de programmes, comme CFML [2] ou Iris [4], en définissant directement la sémantique du langage considéré dans l'assistant de preuve utilisé. L'utilisation de ces outils requiert la définition au préalable de la sémantique du langage, ce qui peut demander beaucoup d'efforts. Une extension possible de notre approche serait de générer une telle description à partir d'une sémantique squelettique.

# 6 Conclusion

Nous avons présenté une formalisation des sémantiques squelettiques, de leur sémantique naturelle, ainsi qu'un outil permettant de générer automatiquement du code Coq utilisable avec cette formalisation à partir d'une description textuelle de la sémantique. Nous avons évalué notre approche grâce à deux applications : l'étude de la sémantique naturelle d'une boucle `while`, et la preuve de correction de compilation d'expressions arithmétiques. Ces travaux ont de nombreuses extensions naturelles. Tout d'abord, nous pourrons définir une sémantique abstraite dans le même cadre et prouver sa correction, comme cela a été fait dans [1]. Ensuite, nous pourrons appliquer notre traducteur à d'autres langages et valider que la formalisation générée est bien utilisable pour prouver des propriétés de programmes. Enfin, nous pourrons définir une interprétation prenant la forme d'une *sémantique axiomatique* pour simplifier les preuves de programmes.

# Références

[1] Martin Bodin, Philippa Gardner, Thomas Jensen, and Alan Schmitt. Skeletal Semantics and their Interpretations. *Proceedings of the ACM on Programming Languages*, 44 :1–31, 2019.

[2] Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 418–430. ACM, 2011.

[3] Nathanaël Courant, Enzo Crance, and Alan Schmitt. Necro : Animating Skeletons. In *ML 2019*, Berlin, Germany, August 2019.

[4] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up : A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28 :e20, 2018.

[5] Gilles Kahn. Natural semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1987.

[6] Liyi Li and Elsa L. Gunter. Isak : A complete semantics of $\mathbb{K}$. Technical report, Computer Science, Univ. of Illinois Urbana-Champaign, 2018.

[7] Grigore Roşu. Matching logic. *Logical Methods in Computer Science*, 13(4) :1–61, 2017.

[8] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the $\mathbb{K}$ semantic framework. *Journal of Logic and Algebraic Programming*, 79(6) :397–434, 2010.

[9] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott : Effective tool support for the working semanticist. *Journal of Functional Programming*, 20(1) :71–122, 2010.