Necro, la sémantique sans y laisser les os : conception d'un système formel de description et de manipulation de sémantiques opérationnelles

Louis Noizet

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES 1

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par
Louis NOIZET

Necro : la sémantique sans y laisser les os

Conception d'un système formel de description et de manipulation de sémantiques opérationnelles

Thèse présentée et soutenue à Rennes, le 29 septembre 2022
Unité de recherche : IRISA

Rapporteurs avant soutenance :

Jean-Christophe FILLIATRE	Directeur de recherche, CNRS
Jean-Bernard STEFANI	Directeur de recherche, Inria Grenoble

Composition du Jury :

Président :	David BAELDE	Maître de conférence, École Normale Supérieure de Rennes
Examinateurices :	Chantal KELLER	Maîtresse de conférence, Université Paris-Saclay
	Assia MAHBOUBI	Directrice de recherche, Inria Nantes
	Yann RÉGIS-GIANAS	Ingénieur, Nomadic Labs
Dir. de thèse :	Alan SCHMITT	Directeur de recherche, Inria Rennes

TABLE DES MATIÈRES

Remerciements

Tout seul on va plus vite, ensemble on va plus loin

Proverbe africain

Fidèle à la coutume, je souhaite commencer cette thèse en remerciant toutes les personnes qui m'ont aidé à arriver au résultat produit ici.

Avant tout, je remercie Alan pour sa patience, son humour, ses bons conseils, et pour avoir réussi à me supporter durant 3 années de thèse malgré une productivité assez inégale.

Je remercie les rapporteur·es et examinateurices, pour avoir accepté de participer au jury de ma thèse, et pour les questions pertinentes qui m'ont ouvert des perspectives de recherche.

Je remercie très cordialement l'équipe Celtique de ces trois dernières années, pour les séminaires enrichissants, pour vos interventions intéressantes et précieuses lors de mes présentations, et pour les bons moments passés autour d'un café, d'un thé, ou d'un chocolat chaud (avant le Covid surtout).

Je souhaite ensuite remercier les intervenant·es des conférences auxquelles j'ai participé, et les reviewers des conférences pour lesquelles j'ai fait une soumission.

Je voudrais remercier spécifiquement Samuel, pour avoir mis Necro sur opam, mais aussi et peut-être surtout, pour m'avoir forcé à m'aérer l'esprit en me traînant en randonnée, quand j'étais dans la détresse émotionnelle.

Merci à Adam, qui m'a recueilli chez lui quand j'ai eu besoin de trouver un nouveau foyer.

Merci à ma famille qui m'a soutenu et aimé, depuis déjà bien longtemps, et qui a toujours essayé de faire ce qui était le meilleur pour moi, avec un succès généralement au rendez-vous.

Merci notamment à Thomas, Santiago et Carybe, qui m'ont donné envie d'aller fréquemment à l'escalade, et m'ont aidé à garder un esprit sain dans un corps (relativement) sain.

Merci aux restaurants de pizza de la ville de Rennes, et principalement Marc, du regretté restaurant Pizz'Agusta, pour m'avoir donné le carburant nécessaire pour affronter ma thèse.

Merci à toutes les personnes qui m'ont guidé et accompagné chez les Scouts et Guides de France. C'est grâce à mes expériences SGDF que j'ai acquis de nombreuses qualités humaines, qui m'ont aidé dans la vie et dans le cadre des études.

Enfin, merci infiniment à Santiago, Jade, Jean-Loup, Roméo, Théo et Vincent pour avoir préparé un pot de thèse très qualitatif, et majoritairement végétalien.

Introduction

[P]ripensu bone antaŭ lego :
tute ne estas mi altruda,
forĵetu min, retiru vin
al via dika ŝel' testuda.

Peter Peneter, *Sekretaj Sonetoj*

Qu'est-ce qu'une sémantique

La sémantique, c'est donner du sens à des mots, à des phrases, à des langages. C'est ce qui nous permet de communiquer de manière relativement fiable.

Si par exemple nous n'utilisons pas tous le mot licorne pour désigner un animal à quatre pattes, similaire au cheval à l'exception près qu'il a une corne au milieu du front, nous risquons un malentendu assez grave. Imaginons un instant que votre voisin·e emploie le mot licorne pour désigner un rhinocéros, vous risquez de ne pas la·e prendre au sérieux si iel vous dit « Attention, une licorne te fonce dessus ! »

Les dictionnaires garantissent une certaine cohésion de la sémantique des mots, mais l'évolution de la langue ainsi que la multiplicité des dictionnaires rend impossible une parfaite unicité de la sémantique [1]. Ainsi, suivant les régions, un·e collègue ne travaille pas nécessairement avec vous, et une poche n'est pas nécessairement attachée à vos vêtements. Cette polysémie de certains mots et de certaines expressions peut entraîner des confusions critiques. Si vous invitez un·e Québécois·e à diner, vous pouvez être surpris·e de la·e voir débarquer chez vous à midi.

De même, si nous nous mettons d'accord sur le sens des mots individuellement, mais pas des phrases construites, vous risquez d'être très surpris·e si on vous dit qu'il pleut des cordes, et vous risquez d'omettre de prendre votre interlocuteurice au sérieux et un parapluie.

Ainsi, lorsque deux personnes communiquent, il est important de s'assurer que la sémantique des mots employés par les deux personnes est non-ambiguë.

Importance des sémantiques formelles

Lorsqu'il s'agit d'un ordinateur, le problème est le même. Si vous communiquez avec votre ordinateur, via un programme informatique écrit dans un langage de programmation quelconque,

1. Par exemple, au moment de la rédaction de cette thèse, le petit Robert admet le mot « iel », alors que la plupart des dictionnaires, notamment celui de l'Académie Française ne le reconnaissent pas encore.

vous souhaitez que votre ordinateur comprenne ce que vous souhaitez faire. Si vous demandez à votre téléphone de vous réveiller à 7h le lendemain, il n'est pas souhaitable que le réveil sonne à 6h, et il est probablement encore moins souhaitable qu'il sonne à 19h (sauf si vous êtes un doctorant en télé-travail, auquel cas personne ne sera jamais au courant que vous avez dormi au lieu de travailler).

Certes, se réveiller trop tôt ou trop tard semble déjà peu souhaitable, mais il est des cas bien plus critiques. Le programme activé par la manette d'arrêt d'urgence d'un métro, le logiciel faisant fonctionner un pacemaker, ou le logiciel faisant fonctionner une centrale nucléaire, sont autant de programmes dont le dysfonctionnement pose un problème d'une gravité importante, en causant la potentielle perte de nombreuses vies humaines.

Afin de garantir que votre programme fait bien ce qu'on attend de lui, il faut donc déjà avoir donné du sens au langage que vous employez. Il faut donc donner une sémantique claire et unique à ce langage. Donner une sémantique à un langage informatique est significativement plus simple que d'en donner une à une langue naturelle pour plusieurs raisons :

— Un langage informatique est généralement sujet à moins d'évolutions qu'une langue naturelle.
— Ces évolutions sont habituellement codifiées et choisie par un organisme en charge du langage (généralement l'organisme qui a initialement crée le langage).
— Un langage informatique utilise peu de « mots », et la grammaire de ces mots est bien plus simple que celle d'une langue usuelle.

Malgré cette simplicité apparente, peu de langages ont une sémantique définie. Le plus souvent, le langage est considéré comme suffisamment explicite pour se passer de sémantique, et parfois, un interpréteur sert de référence. C'est-à-dire que pour savoir ce que signifie un code, il faut l'exécuter et regarder ce qui se passe.

Le manque de définition formelle du langage pose un souci évident, si un code est écrit dans un langage mal spécifié, il est possible que le code ne soit pas adapté à l'implémentation effective de l'interpréteur ou du compilateur utilisé. Mais il y a un deuxième problème plus subtil. Si l'on écrit un code correct pour un interpréteur, qui fait bien ce qui est attendu de lui et est parfaitement sûr, il ne le sera peut-être pas s'il est exécuté avec un autre interpréteur puisque, la sémantique n'étant pas claire, plusieurs interpréteurs ou compilateurs peuvent comprendre le langage de manière différente. Ainsi la sémantique du code écrit dans le langage source, et la sémantique du code traduit par le compilateur ne sont pas nécessairement cohérente.

Comment définir des sémantiques

Maintenant que le·a lecteurice est convaincu·e de l'importance de spécifier la sémantique du langage qu'on utilise, il s'agit de voir comment se réalise cette spécification.

Certaines spécifications sont écrites dans un langage humain. C'est par exemple le cas de la spécification de JavaScript avec ECMA-262. Cette sémantique permet de pouvoir s'assurer du

sens d'un programme, mais elle n'est pas mécanisée, c'est-à-dire qu'elle n'est pas écrite dans un langage compris par un ordinateur, ce qui implique plusieurs limitations :

— Il peut y avoir des erreurs dans la sémantique, et il est potentiellement difficile de les détecter.
— Il y a généralement des assertions non vérifiées (ECMA-262 fait régulièrement des assertions [2] qui sont supposées vraies par construction).
— On ne peut pas exécuter un code à partir de la spécification.
— Il est impossible de faire des preuves sur ordinateur de la correction d'un programme.

D'autres spécifications sont informatisées, pour pouvoir avoir des exécutions prouvées correctes, et pour pouvoir prouver informatiquement certaines propriétés de code. C'est le cas par exemple de JSCert qui propose une spécification de JavaScript écrite à l'aide de l'assistant de preuve Coq.

Plusieurs outils existent pour définir une sémantique de manière informatique. Le plus standard et le plus fréquent est d'utiliser un assistant de preuve comme Coq ou Isabelle/HOL. Ces sémantiques ne sont généralement pas exécutables, et doivent donc être extraites vers un autre langage pour les exécuter (le système d'extraction de Coq permet par exemple de générer du OCaml). Cette méthode est relativement efficace, mais les changements de choix de conception, comme passer d'un plongement superficiel à un plongement profond, ont généralement un coût important.

On préfèrera donc généralement un langage de spécification, qui proposera ensuite d'extraire vers un assistant de preuve, vers un langage exécutable, et vers d'autres outils si besoin (un débogueur par exemple). Des outils existants sont Lem, Ott et K qui ont chacun leur limitations, et que nous présentons dans la section 1.3.

Cette thèse présente le langage Skel, un langage de spécification, à la fois léger et puissant, ainsi que l'écosystème Necro qui permet de manipuler les spécifications, pour les extraire vers divers outils, ou pour les transformer directement.

La mise à jour du langage Skel ainsi que l'essentiel de l'écosystème Necro ont été créés pendant cette thèse, donc sur une durée de 3 ans par un doctorant. C'est une démonstration de la simplicité de Skel que de voir à quelle vitesse il a pu donner lieu à une variété d'outils, et la facilité d'en créer de nouveaux.

Structure de la thèse

Ce rapport de thèse est constitué de 7 parties. Dans la section 1, on replace dans leur contexte les sémantiques, et en particulier les sémantiques opérationnelles à plutôt grand pas. La deuxième section décrit le fonctionnement du langage Skel, dont la version actuelle est une contribution de cette thèse. Les sections 3, 4 et 5 présentent respectivement les outils Necro Lib, Necro ML et Necro Coq, qui ont été construits dans leur version actuelle dans le cadre de cette thèse. Enfin,

2. https://262.ecma-international.org/12.0/#assert

la section 6 présente d'autres applications qui ont été faite de Skel et Necro, dont Necro Debug, qui est une contribution de cette thèse.

Implémentation

Necro Lib, qui a été presque intégralement réalisé dans le cadre de cette thèse, totalise plus de 5000 lignes de codes. Necro ML représente plus de 1300 lignes de code, et plus de 1000 lignes de test. Necro Coq quant à lui contient peu de lignes de code (un peu plus de 300), ce qui prouve la facilité de créer un outil. Les fichiers Coq nécessaires pour faire fonctionner les sémantiques traduites par Necro Coq font un peu plus de 2000 lignes. Les preuves réalisées vis-à-vis de ces fichiers dans le cadre de cette thèse totalisent 3000 lignes supplémentaires. Et les tests réalisés durant cette thèse font encore 800 lignes de plus. Necro Debug enfin, totalise 1800 lignes de code.

Au total, ce sont plus de 10 000 lignes de code qui ont été écrites dans le cadre de cette thèse. Naturellement, ces lignes ne sont que le résultat final d'un travail de recherche de 3 ans, qui a amené d'une part à de nombreuses évolutions dans le langage Skel et donc dans les outils Necro, et d'autre part, à des travaux d'améliorations des outils, indépendamment du langage en soi, en ce qui concerne l'expérience utilisateurice, mais aussi la modularité et la lisibilité de ces outils.

Les travaux réalisés ont par ailleurs mené à plusieurs participations à des conférences. Nous avons publié deux articles dans le cadre des Journées Francophones des Langages Applicatifs en 2020 et 2021 [20, 13], et un article intitulé « Stating and Handling Semantics » qui sera présenté à ICTCS 2022.

Enfin, en ce qui concerne la facilité d'utilisation de ces outils, ils ont été rendu disponibles sur un dépôt opam non-officiel par Samuel Risbourg, il est possible de tout installer pour pouvoir faire des essais en parallèle de la lecture de cette thèse en exécutant les commandes suivantes :

```
opam update
opam switch create necro 4.14.0
opam switch necro
eval $(opam env)
opam repository add necro https://gitlab.inria.fr/skeletons/opam-repository.git#necro
opam install -y necrolib necroml necrocoq necrodebug
```

Chapitre 1

Contexte

That's one small step for man, one giant leap for mankind.

Neil Armstrong

1.1 Définitions

Pour pouvoir raisonner sur des programmes, et prouver des propriétés sur ceux-ci, il est essentiel de pouvoir donner la sémantique dudit langage. Nous illustrerons cette section en prenant un exemple simple, le langage IMP. Par la suite, nous utiliserons surtout le λ-calcul comme support, mais l'exemple de ce langage impératif est plus complet et permet donc de montrer plus de notions de manière immédiate, et de mieux illustrer la différence entre les choix de sémantique. La grammaire du langage IMP est définie en figure 1.1. On marque d'un point les opérateurs d'addition, négation et égalité ainsi que les constantes entières et booléennes, afin de les différencier des opérations sur les entiers et booléens et des entiers et booléens eux-mêmes. En effet, 8 est un entier et $+$ est l'addition sur ces entiers, alors que $\dot{8}$ et $\dot{+}$ ne sont que des constructions syntaxiques qui servent à écrire une expression IMP.

Donner la sémantique d'un langage consiste à expliquer comment on peut interpréter une construction du langage, et lui donner du sens, vis-à-vis d'un résultat attndu. Le type du résultat attendu est défini spécifiquement pour chaque langage. Par exemple, pour le langage IMP, les expressions sont évaluées vers des valeurs, une valeur étant un entier ou un booléen, tandis que les instructions sont évaluées vers des états mémoires.

Il existe au moins trois manières principales de donner la sémantique d'un langage.

— La sémantique dénotationnelle donne à chaque programme une dénotation. Cette dénotation est généralement une fonction qui prend en argument une entrée (dans le cas de

$$\begin{array}{lll} \text{EXPR} & e ::= & \dot{n} \mid \dot{\bot} \mid \dot{\top} \mid x \mid e \dot{+} e \mid e \dot{=} e \mid \dot{\neg} e \\ \text{STMT} & s ::= & \mathit{skip} \mid x := e \mid s;\ s \mid \mathit{if}\ e\ s\ s \mid \mathit{while}\ e\ s \end{array}$$

Figure 1.1 – Grammaire du langage IMP.

$$\frac{\{P \wedge b\} s \{P\}}{\{P\} \mathit{while}\, b\, s \{P \wedge \neg b\}}$$

FIGURE 1.2 – La règle pour la sémantique axiomatique d'un bloc *while*.

$$\frac{\sigma, e \Downarrow_e \top \qquad \sigma, s_1 \Downarrow_s \sigma'}{\sigma, \mathit{if}\, e\, s_1\, s_2 \Downarrow_s \sigma'} \qquad \frac{\sigma, e \Downarrow_e \bot \qquad \sigma, s_2 \Downarrow_s \sigma'}{\sigma, \mathit{if}\, e\, s_1\, s_2 \Downarrow_s \sigma'}$$

FIGURE 1.3 – Règles pour la sémantique opérationnelle à grand pas d'un bloc *if*.

IMP, l'état de la mémoire avant l'exécution), et renvoie une sortie (pour EXPR, il s'agira d'une valeur, pour STMT d'un nouvel état mémoire).

— La sémantique axiomatique se base sur la logique de Hoare pour prouver des triplets de Hoare de la forme $\{P\}s\{Q\}$. Ces triplets garantissent que l'exécution du programme s à partir d'un état mémoire respectant P donnera un résultat respectant Q. Elle demande de « deviner » les propositions qui seront pertinentes pour prouver le résultat final. Par exemple, pour un bloc *while*, il faudra trouver l'invariant P (voir figure 1.2).

— La sémantique opérationnelle quant à elle, se rapproche plus de la manière dont un ordinateur exécute un programme. Étant donnés un état et une expression, elle donne le résultat, généralement avec une définition inductive qui prendra en argument les différentes étapes de l'exécution.

Dans la suite nous nous centrerons sur la sémantique opérationnelle. Il existe plusieurs manières de donner la sémantique opérationnelle d'un langage. La sémantique à petit pas explique comment se fait un unique pas de calcul, et on peut l'itérer pour arriver au résultat. La sémantique à grand pas donne directement le résultat, en utilisant des règles inductives.

1.1.1 Grand pas

Une sémantique grand pas définit inductivement les calculs nécessaires pour arriver à un résultat. Elle prend donc en argument une expression, et souvent un contexte ou un état, et elle renvoie directement une valeur. On donne en figure 1.3 les deux règles de la sémantique grand pas d'un *if*.

On voit que la règle fait tout d'un coup. Elle calcule la condition, sélectionne la bonne instruction, et l'évalue. On donne toutes les règles de la sémantique à grand pas de IMP en figure 1.4.

$$\frac{}{\sigma, \dot{n} \Downarrow_e n}\ \textsc{Const} \qquad \frac{}{\sigma, \dot{\top} \Downarrow_e \top}\ \textsc{Top} \qquad \frac{}{\sigma, \dot{\bot} \Downarrow_e \bot}\ \textsc{Bot} \qquad \frac{\sigma(x) = v}{\sigma, x \Downarrow_e v}\ \textsc{Var}$$

$$\frac{\sigma, e_1 \Downarrow_e n_1 \qquad \sigma, e_2 \Downarrow_e n_2 \qquad n_1 + n_2 = n}{\sigma, e_1 \dot{+} e_2 \Downarrow_e n}\ \textsc{Add} \qquad \frac{\sigma, e_1 \Downarrow_e n \qquad \sigma, e_2 \Downarrow_e n}{\sigma, e_1 \dot{=} e_2 \Downarrow_e \top}\ \textsc{EqTrue}$$

$$\frac{\sigma, e_1 \Downarrow_e n_1 \qquad \sigma, e_2 \Downarrow_e n_2 \qquad n_1 \neq n_2}{\sigma, e_1 \dot{=} e_2 \Downarrow_e \bot}\ \textsc{EqFalse} \qquad \frac{\sigma, e \Downarrow_e b}{\sigma, \dot{\neg} e \Downarrow_e \neg b}\ \textsc{Neg}$$

$$\frac{}{\sigma, \mathit{skip} \Downarrow_s \sigma}\ \textsc{Skip} \qquad \frac{\sigma, e \Downarrow_e v}{\sigma, x := e \Downarrow_s \sigma[x \leftarrow v]}\ \textsc{Asn} \qquad \frac{\sigma, s_1 \Downarrow_s \sigma' \qquad \sigma', s_2 \Downarrow_s \sigma''}{\sigma, (s_1;\ s_2) \Downarrow_s \sigma''}\ \textsc{Seq}$$

$$\frac{\sigma, e \Downarrow_e \top \qquad \sigma, s_1 \Downarrow_s \sigma'}{\sigma, (\mathit{if}\ e\ s_1\ s_2) \Downarrow_s \sigma'}\ \textsc{IfTrue} \qquad \frac{\sigma, e \Downarrow_e \bot \qquad \sigma, s_2 \Downarrow_s \sigma'}{\sigma, (\mathit{if}\ e\ s_1\ s_2) \Downarrow_s \sigma'}\ \textsc{IfFalse}$$

$$\frac{\sigma, e \Downarrow_e \top \qquad \sigma, s \Downarrow_s \sigma' \qquad \sigma', (\mathit{while}\ e\ s) \Downarrow_s \sigma''}{\sigma, (\mathit{while}\ e\ s) \Downarrow_s \sigma''}\ \textsc{WhileTrue} \qquad \frac{\sigma, e \Downarrow_e \bot}{\sigma, (\mathit{while}\ e\ s) \Downarrow_s \sigma}\ \textsc{WhileFalse}$$

FIGURE 1.4 – Règles pour la sémantique à grand pas de IMP.

Voici ci-dessous les règles d'inférence associées à l'exécution de la formule $x\dot{+}(y\dot{+}\dot{2})$ dans l'état σ, où x est associé à 1, et y à 3 :

$$\frac{\dfrac{\sigma(x) = 1}{\sigma, x \Downarrow_e 1}\ \textsc{Var} \qquad \dfrac{\dfrac{\sigma(y) = 3}{\sigma, y \Downarrow_e 3}\ \textsc{Var} \qquad \dfrac{}{\sigma, \dot{2} \Downarrow_e 2}\ \textsc{Const} \qquad 3 + 2 = 5}{\sigma, y\dot{+}\dot{2} \Downarrow_e 5}\ \textsc{Add} \qquad 1 + 5 = 6}{\sigma, x\dot{+}(y\dot{+}\dot{2}) \Downarrow_e 6}\ \textsc{Add}$$

1.1.2 Petit pas

Dans la sémantique à petit pas par contre, on décompose les étapes une par une. On travaille donc avec des configurations. On calcule une étape et on renvoie une nouvelle configuration, obtenue après un pas de calcul. Pour pouvoir reconstruire une configuration après chaque étape, on doit considérer différentes sortes de configurations. Quand le calcul est terminé, on utilise une configuration contenant uniquement une valeur. Quand le calcul commence, on utilise un état et l'expression ou instruction à évaluer. Mais pour certains constructeurs, on est obligés d'ajouter des nouveaux constructeurs pour stocker des données intermédiaires. Par exemple, pour un +, on a besoin de stocker la valeur de la première expression quand elle est calculée. On commence donc avec la somme de deux expressions, avec le constructeur initial $\dot{+}$, de signature

$$\begin{array}{lll} \textsc{expr} & e ::= & \dot{n} \mid \dot{\bot} \mid \dot{\top} \mid x \mid e \dot{+} e \mid v \ddot{+} e \mid e \dot{=} e \mid v \ddot{=} e \mid \dot{\neg} e \\ \textsc{stmt} & s ::= & \mathit{skip} \mid x := e \mid s;\ s \mid \mathit{if}\, e\, s\, s \mid \mathit{while}\, e\, s \end{array}$$

FIGURE 1.5 – Grammaire du langage IMP.

$\texttt{expr} * \texttt{expr} \to \texttt{expr}$. Puis, lorsque la première expression est évaluée vers une valeur entière, on utilise le constructeur $\ddot{+}$, de signature $\texttt{int} * \texttt{expr} \to \texttt{expr}$. Enfin, lorsque le calcul des deux expressions est terminé, on les additionne avec l'addition sur les entiers. On donne la grammaire étendue en figure 1.5.

On donne toutes les règles de la sémantique à petit pas de IMP en figure 1.6, et on développe ci-dessous les trois étapes de l'évaluation de $x\dot{+}(y\dot{+}\dot{2})$ dans le même état σ que précédemment :

$$\dfrac{\dfrac{\dfrac{\sigma(x) = 1}{\sigma, x \longrightarrow_e 1}\ \textsc{Var}}{\sigma, x\dot{+}(y\dot{+}\dot{2}) \longrightarrow_e \sigma, 1\ddot{+}(y\dot{+}\dot{2})}\ \textsc{AddLRet}}$$

$$\dfrac{\dfrac{\dfrac{\sigma, y \longrightarrow_e 3}{\sigma, y\dot{+}\dot{2} \longrightarrow_e \sigma, 3\ddot{+}\dot{2}}\ \textsc{AddLRet}}{\sigma, 1\ddot{+}(y\dot{+}\dot{2}) \longrightarrow_e \sigma, 1\ddot{+}(3\ddot{+}\dot{2})}\ \textsc{AddRCont}}$$

$$\textsc{AddRRet}\ \dfrac{\textsc{AddRRet}\ \dfrac{\sigma, \dot{2} \longrightarrow_e 2}{\sigma, (3\ddot{+}\dot{2}) \longrightarrow_e 5}}{\sigma, 1\ddot{+}(3\ddot{+}\dot{2}) \longrightarrow_e 6}$$

On a donc :

$$\begin{array}{rcl} \sigma, x\dot{+}(y\dot{+}\dot{2}) & \longrightarrow_e & \sigma, 1\ddot{+}(y\dot{+}\dot{2}) \\ & \longrightarrow_e & \sigma, 1\ddot{+}(3\ddot{+}\dot{2}) \\ & \longrightarrow_e & 6 \end{array}$$

1.1.3 Machine abstraite

Pour éviter le problème d'avoir à reconstruire des termes, on peut utiliser une machine abstraite. Une machine abstraite est une méthode d'évaluation où on sélectionne la partie du terme à évaluer, et on garde le contexte en mémoire pour plus tard. Ainsi, une configuration d'évaluation est généralement constituée d'un état, d'un terme à évaluer, et d'un contexte. On donne toutes les règles de la sémantique à machine abstraite de IMP en figure 1.7.

$$\frac{}{\sigma, \dot{n} \longrightarrow_e n}\ \textsc{Const} \qquad \frac{}{\sigma, \dot{\top} \longrightarrow_e \top}\ \textsc{Top} \qquad \frac{}{\sigma, \dot{\bot} \longrightarrow_e \bot}\ \textsc{Bot} \qquad \frac{\sigma(x) = v}{\sigma, x \longrightarrow_e v}\ \textsc{Var}$$

$$\frac{\sigma, e_1 \longrightarrow_e \_, e_1'}{\sigma, e_1 \dot{+} e_2 \longrightarrow_e \sigma, e_1' \dot{+} e_2}\ \textsc{AddLCont} \qquad \frac{\sigma, e_1 \longrightarrow_e n_1}{\sigma, e_1 \dot{+} e_2 \longrightarrow_e \sigma, n_1 \ddot{+} e_2}\ \textsc{AddLRet}$$

$$\frac{\sigma, e_2 \longrightarrow_e \_, e_2'}{\sigma, n_1 \ddot{+} e_2 \longrightarrow_e \sigma, n_1 \ddot{+} e_2'}\ \textsc{AddRCont} \qquad \frac{\sigma, e_2 \longrightarrow_e n_2}{\sigma, n_1 \ddot{+} e_2 \longrightarrow_e (n_1 + n_2)}\ \textsc{AddRRet}$$

$$\frac{\sigma, e_1 \longrightarrow_e \_, e_1'}{\sigma, e_1 \dot{=} e_2 \longrightarrow_e \sigma, e_1' \dot{=} e_2}\ \textsc{EqLCont} \qquad \frac{\sigma, e_1 \longrightarrow_e n_1}{\sigma, e_1 \dot{=} e_2 \longrightarrow_e \sigma, n_1 \ddot{=} e_2}\ \textsc{EqLRet}$$

$$\frac{\sigma, e_2 \longrightarrow_e \_, e_2'}{\sigma, n_1 \ddot{=} e_2 \longrightarrow_e \sigma, n_1 \ddot{=} e_2'}\ \textsc{EqRCont} \qquad \frac{\sigma, e_2 \longrightarrow_e n}{\sigma, n \ddot{=} e_2 \longrightarrow_e \top}\ \textsc{EqRRetTrue}$$

$$\frac{\sigma, e_2 \longrightarrow_e n_2 \qquad n_2 \neq n_1}{\sigma, n_1 \ddot{=} e_2 \longrightarrow_e \bot}\ \textsc{EqRRetFalse} \qquad \frac{\sigma, e \longrightarrow_e \sigma, e'}{\sigma, \dot{\neg} e \longrightarrow_e \sigma, \dot{\neg} e'}\ \textsc{NegCont}$$

$$\frac{\sigma, e \longrightarrow_e b}{\sigma, \dot{\neg} e \longrightarrow_e \neg b}\ \textsc{NegRet}$$

$$\frac{}{\sigma, \mathit{skip} \longrightarrow_s \sigma}\ \textsc{Skip} \qquad \frac{}{\sigma, x := e \longrightarrow_s (\sigma[x \leftarrow e])}\ \textsc{Asn}$$

$$\frac{\sigma, s_1 \longrightarrow_s \sigma', s_1'}{\sigma, (s_1;\ s_2) \longrightarrow_s \sigma', (s_1';\ s_2)}\ \textsc{SeqLCont} \qquad \frac{\sigma, s_1 \longrightarrow_s \sigma'}{\sigma, (s_1;\ s_2) \longrightarrow_s \sigma', s_2}\ \textsc{SeqLRet}$$

$$\frac{\sigma, e \longrightarrow_e \_, e'}{\sigma, (\mathit{if}\ e\ s_1\ s_2) \longrightarrow_s \sigma, (\mathit{if}\ e'\ s_1\ s_2)}\ \textsc{IfLCont} \qquad \frac{\sigma, e \longrightarrow_e \top}{\sigma, (\mathit{if}\ e\ s_1\ s_2) \longrightarrow_s \sigma, s_1}\ \textsc{IfLRetTrue}$$

$$\frac{\sigma, e \longrightarrow_e \bot}{\sigma, (\mathit{if}\ e\ s_1\ s_2) \longrightarrow_s \sigma, s_2}\ \textsc{IfLRetFalse} \qquad \frac{}{\sigma, (\mathit{while}\ e\ s) \longrightarrow_s \sigma, \mathit{if}\ e\ (s;\ \mathit{while}\ e\ s)\ \mathit{skip}}\ \textsc{While}$$

Figure 1.6 – Règles pour la sémantique à petit pas de IMP.

Il y a deux modes. Le mode d'évaluation, représenté avec des chevrons, où une expression est en cours d'évaluation, et le mode continuation, représenté avec des crochets, où l'on a déjà une valeur, et où on dépile le contexte.

On montre comment calculer la même expression $x + (y + 2)$ dans l'état σ. On a donc :

$$\begin{aligned}
\langle x + (y + 2), \sigma, \emptyset \rangle &\rightarrow \langle x, \sigma, (\sigma, \square + (y + 2)) :: \emptyset \rangle \\
&\rightarrow [1, (\sigma, \square + (y + 2)) :: \emptyset] \\
&\rightarrow \langle (y + 2), \sigma, (1 + \square) :: \emptyset \rangle \\
&\rightarrow \langle y, \sigma, (\sigma, \square + 2) :: (1 + \square) :: \emptyset \rangle \\
&\rightarrow [3, (\sigma, \square + 2) :: (1 + \square) :: \emptyset] \\
&\rightarrow \langle 2, \sigma, (3 + \square) :: (1 + \square) :: \emptyset \rangle \\
&\rightarrow [2, (3 + \square) :: (1 + \square) :: \emptyset] \\
&\rightarrow [5, (1 + \square) :: \emptyset] \\
&\rightarrow [6, \emptyset]
\end{aligned}$$

1.2 Équivalence

Ces différentes méthodes pour décrire la sémantique d'un langage ont la même puissance, et sont généralement équivalentes. Ainsi, si l'on considère les trois sémantiques données dans les sections 1.1.1, 1.1.2 et 1.1.3, on a bien les résultats suivants, que nous ne prouverons pas :

Theorem 1. *Les trois assertions suivantes sont équivalentes*

- $\sigma, e \Downarrow_e v$.
- $\sigma, e \longrightarrow_e^* v$.
- $\langle e, \sigma, \emptyset \rangle \rightarrow^* [v, \emptyset]$.

Theorem 2. *Les trois assertions suivantes sont équivalentes*

- $\sigma, s \Downarrow_s \sigma'$.
- $\sigma, s \longrightarrow_s^* \sigma'$.
- $\langle s, \sigma, \emptyset \rangle \rightarrow^* [\sigma', \emptyset]$.

1.3 Langages de description de sémantiques

Le rôle de cette thèse est de présenter un langage de description de sémantiques, Skel, et l'écosystème Necro qui permet de le manipuler. Plusieurs langages de description de sémantiques existent déjà, et l'auteur de cette thèse ni son directeur de thèse n'ont l'hybris de prétendre que leur langage est meilleur ou mieux pensé. Nous expliquerons cependant pourquoi l'usage que nous

$$
\begin{aligned}
\langle n, \sigma, k\rangle &\to [n, k]\,, n \in \mathbb{N} \\
\langle b, \sigma, k\rangle &\to [b, k]\,, b \in \{\top, \bot\} \\
\langle x, \sigma, k\rangle &\to [\sigma(x), k] \\
\langle e_1 + e_2, \sigma, k\rangle &\to \langle e_1, \sigma, (\sigma, \square + e_2) :: k\rangle \\
\langle e_1 = e_2, \sigma, k\rangle &\to \langle e_1, \sigma, (\sigma, \square = e_2) :: k\rangle \\
\langle \neg e, \sigma, k\rangle &\to \langle e, \sigma, (\neg\square) :: k\rangle \\
\langle \mathit{skip}, \sigma, k\rangle &\to [\sigma, k] \\
\langle x := e, \sigma, k\rangle &\to \langle e, \sigma, (\sigma, x := \square) :: k\rangle \\
\langle s_1;\ s_2, \sigma, k\rangle &\to \langle s_1, \sigma, (\square;\ s_2) :: k\rangle \\
\langle \mathit{if}\, e\, s_1\, s_2, \sigma, k\rangle &\to \langle e, \sigma, (\sigma, \mathit{if}\, \square\, s_1\, s_2) :: k\rangle \\
\langle \mathit{while}\, e\, s, \sigma, k\rangle &\to \langle e, \sigma, (\sigma, \mathit{while}\, \square_e\, s) :: k\rangle \\
[m, (\sigma, \square + e_2) :: k] &\to \langle e_2, \sigma, (m + \square) :: k\rangle \\
[n, (m + \square) :: k] &\to [m + n, k] \\
[m, (\sigma, \square = e_2) :: k] &\to \langle e_2, \sigma, (m = \square) :: k\rangle \\
[n, (m = \square) :: k] &\to [\top, k]\,, m = n \\
[n, (m = \square) :: k] &\to [\bot, k]\,, m \neq n \\
[\top, (\neg\square) :: k] &\to [\bot, k] \\
[\bot, (\neg\square) :: k] &\to [\top, k] \\
[v, (\sigma, x := \square) :: k] &\to [\sigma[x \leftarrow v], k] \\
[\sigma, (\square;\ s_2) :: k] &\to \langle s_2, \sigma, k\rangle \\
[\top, (\sigma, \mathit{if}\, \square\, s_1\, s_2) :: k] &\to \langle s_1, \sigma, k\rangle \\
[\bot, (\sigma, \mathit{if}\, \square\, s_1\, s_2) :: k] &\to \langle s_2, \sigma, k\rangle \\
[\top, (\sigma, \mathit{while}\, \square_e\, s) :: k] &\to \langle s, \sigma, (\square;\ \mathit{while}\, e\, s) :: k\rangle \\
[\bot, (\sigma, \mathit{while}\, \square_e\, s) :: k] &\to [\sigma, k]
\end{aligned}
$$

FIGURE 1.7 – Règles pour la sémantique à machine abstraite de IMP.

envisageons d'un langage de spécification rend Skel et Necro plus pertinents que les précédents outils. Pour présenter les outils, nous utiliserons le λ-calcul en appel par valeur. IMP étant un langage assez gros, cet exemple est plus minimal et fonctionnel.

Attendus

Pour être utilisable et maintenable, on attend plusieurs choses d'un langage de spécification. Tout d'abord, le plus évident et le plus nécessaire, est qu'il doit permettre de spécifier une sémantique de manière lisible, compréhensible, et maintenable.

On attend également que le langage soit simple et ait une sémantique claire et bien définie. Enfin, on attend que le langage dispose d'un ensemble d'outils pour l'exécuter ou l'injecter dans des assistants de preuves. Il faut idéalement qu'il ait une interface suffisamment simple pour pouvoir facilement programmer d'autres outils.

Un bon langage de spécification peut cependant avoir une méta-sémantique configurable, de manière à pouvoir donner des sens différents à une même sémantique écrite dans ce langage. On attend alors que les différentes interprétations possibles soient clairement documentées.

Certains langages de spécifications proposent également une gestion de types natifs, ou une gestion native des lieurs pour simplifier l'écriture de sémantiques. Ces ajouts peuvent néanmoins complexifier la manipulation des sémantiques, et il faut donc choisir entre simplicité d'écriture et simplicité de manipulation.

Enfin, la possibilité de sous-spécification est utile, car elle permet de décrire une sémantique de manière incrémentale, et elle permet de laisser certaines décisions libres suivant les choix d'implémentation.

Skel et Necro

D'abord introduit par Bodin *et al.* [5], le langage Skel et l'outil Necro valident toutes ces propriétés, comme on le montrera dans cette thèse. En particulier, sa minimalité permet à n'importe qui de programmer très simplement un nouvel outil pour exploiter les sémantiques écrites dans ce langage.

Le but de cette section étant de parler des langages et outils préexistants à Skel, nous allons nous concentrer sur ces derniers.

Plongement dans un langage existant

La première méthode et sans doute la plus simple, que l'on utilise souvent pour des langages très simples comme le λ-calcul, est d'utiliser un langage existant comme OCaml ou Haskell, qu'on appelle alors le langage hôte, pour définir la sémantique du nouveau langage. On peut alors utiliser les fonctionnalités du langages hôte pour modéliser les fonctionnalités du langage cible, dans le cadre d'un plongement superficiel, mais on peut aussi les utiliser pour donner de la modularité ou de la flexibilité au langage dans le cadre d'un plongement profond.

Les travaux réalisés par Liang *et al.* [14] montrent comment l'utilisation de transformateurs de monades en Haskell permettent de définir la sémantique d'un langage en utilisant Haskell comme langage hôte, de manière modulaire. De même, Wadler [28] montre comment l'utilisation de monades pour un interpréteur permet de rajouter des fonctionnalités sans avoir à altérer la totalité de l'interpréteur à chaque fois.

On peut aussi décrire une sémantique dans un assistant de preuve, comme JSCert [3], qui a été plongée directement dans l'assistant de preuve Coq. Cela permet d'avoir une sémantique utilisable pour faire directement des preuves formelles, et le mécanisme d'extraction de Coq permet ensuite d'avoir une sémantique OCaml exécutable.

Cependant, il peut être difficile de manipuler une telle formalisation, puisque l'AST de ces langages hôtes n'a pas été conçu dans cette optique. De plus, le coût est élevé pour modifier des choix de conception, comme passer d'un plongement superficiel à un plongement profond.

Skel permet également l'usage de monades, comme nous le montrerons dans la section 2.5. Ces monades assurent en effet une grande flexibilité, notamment dans l'ajout incrémental de fonctionnalités, comme cela a pu être expérimenté dans le cadre de JSkel [12].

Écosystèmes complets

Plusieurs systèmes existent qui fournissent à la fois un langage et un ensemble d'outils pour manipuler ce langage, comme nous le faisons avec Skel et Necro. Cependant, dans les langages que nous présentons ci-dessous, même si la liste d'outils et d'extraction possible est longue, elle est généralement difficile à augmenter. Si l'on souhaite extraire du Lem vers quelque chose qui n'est pas prévu, il y aura un travail non négligeable pour créer un outil. Et en général, l'écosystème n'est pas fait pour être étendu, il faut donc par exemple forker l'outil existant.

Lem

Lem [17, 22] est un langage de spécification qui permet des extractions vers de nombreux outils, notamment un interpréteur OCaml et une formalisation dans les assistants de preuve Isabelle/HOL et Coq. Il ne se réduit pas exclusivement aux langages de programmation, puisqu'il a été utilisé notamment pour spécifier des protocoles de communications, ou un modèle de mémoire faible par exemple.

Lem gère de manière native les ensembles, l'inférence de type et les modules imbriqués. Si cela semble pratique, cela a cependant l'inconvénient de rendre le langage plus lourd et donc plus difficile à manipuler pour créer un nouveau back-end. Il y a notamment dans Lem 4 types d'entiers définis par défaut.

De plus, lors de l'extraction vers Coq, les fonctions sont compilées par des fonctions et les relations par des relations. Cela diffère de notre outil qui ne fait pas la distinction et considère tout comme des relations. Encore une fois, on peut trouver cela utile, mais cela signifie qu'il faut prouver la terminaison des fonctions. Une partie est prouvée automatiquement, mais des preuves

de terminaison peuvent être déférées.

Ensuite, Lem a fait le choix de confondre Prop et bool, et donc de travailler en logique classique dans Coq, donc de perdre en calculabilité. L'argumentation soutenue dans [17] est la suivante :

> The admission of classical axioms to collapse Prop into bool is a matter of taste. Some large Coq developments happily assume classical axioms, others stay firmly within the existing constructive logic provided by Coq. We feel, however, that not restricting the Lem source language to accommodate every nuance exhibited by the backends is worth the admission of these axioms, though to what extent they affect the computational behaviour and automation and proof search tactics of Coq will require further experimentation to fully resolve.

Lem propose également une libraire standard assez complète et autorise à compiler un type/terme vers un type/terme défini par défaut dans le langage cible.

La figure 1.8 présente un λ-calcul par valeur en Lem. Le TEX est généré par Lem.

Ott

Ott [26, 21] est un langage et un outil pour spécifier des langages de programmation. Il permet de générer divers codes, notamment une version LaTeX pour affichage, des versions Coq et Isabelle/HOL pour faire des preuves formelles, une version OCaml pour effectuer des calculs, et une version Lem.

Ott est conçu pour que les codes écrit en Ott ressemblent à des règles d'inférence écrites à la main. Il y a également une gestion native des lieurs qui présente une importance forte pour les auteurices de l'outil et qui est donc très aboutie. Ott gère les listes et les compréhensions de manière interne, avec des syntaxes utilisant des « ... ».

On donne dans la figure 1.9 un λ-calcul avec évaluation par valeur en Ott.

$\mathbb{K}$

$\mathbb{K}$ [24] est un framework exécutable de sémantiques, basé sur des règles de réécriture. Il fonctionne à base de configurations qui capturent les éléments nécessaires au calcul, comme l'état par exemple, tout en cloisonnant les différents éléments, de manière à modulariser au maximum le calcul.

Si la théorie derrière $\mathbb{K}$, et ses nombreuses applications, sont intéressantes, le fort coût d'apprentissage peut être décourageant. De plus, il ne permet pas de s'intéresser aux méta-théories d'un langage [1].

Enfin, la complexité de $\mathbb{K}$ rend difficile d'ajouter des extracteurs, par exemple si l'on souhaite générer du code Coq, ce que $\mathbb{K}$ ne permet pas encore.

On donne dans la figure 1.10 un λ-calcul avec évaluation par valeur en $\mathbb{K}$.

1. https://sympa.inria.fr/sympa/arc/coq-club/2020-02/msg00066.html

```
open import Pervasives

type IDENT = STRING

type LAMBDA =
 | IDENT of IDENT
 | APP of LAMBDA * LAMBDA
 | LAM of IDENT * LAMBDA

let is_value (l : LAMBDA) : BOOL =
   match l with
 | App _ _ → false
 | _ → true
  end

let rec subst (v : LAMBDA) (x : IDENT) (w : LAMBDA) =
  match v with
 | Ident y → if y=x then w else v
 | App v1 v2 → App (subst v1 x w) (subst v2 x w)
 | Lam y body →
   if y=x then Lam y body
   else Lam y (subst body x w)
  end

indreln [Lambda_ss : LAMBDA → LAMBDA → BOOL]

 Lambda_app_ctx_right : ∀ v v' f .
 (Lambda_ss v v')
  ⟹
 Lambda_ss (App f v) (App f v')

 and
 Lambda_app_ctx_left : ∀ v f f' .
 (Lambda_ss f f') ∧
 (is_value v)
  ⟹
 Lambda_ss (App f v) (App f' v)

 and
 Lambda_app_beta_red : ∀ x v w .
 (is_value v)
  ⟹
 Lambda_ss (App (Lam x v) w) (subst v x w)
```

FIGURE 1.8 – λ-calcul avec appel par valeur en Lem.

```
metavar termvar, x ::= {{ com term variable }}
{{ isa string}} {{ coq nat}} {{ hol string}} {{ coq-equality }}
{{ ocaml int}} {{ lex alphanum}} {{ tex \mathit{[[termvar]]} }}

grammar
t :: 't_' ::=                {{ com term }}
| x            :: :: Var     {{ com variable}}
| \ x . t      :: :: Lam (+ bind x in t +) {{ com lambda }}
| t t'         :: :: App     {{ com app }}
| ( t )        :: M :: Paren {{ icho [[t]] }}
| { t / x } t' :: M :: Tsub  {{ icho (tsubst_t [[t]] [[x]] [[t']])}}

v :: 'v_' ::=           {{ com value }}
| \ x . t :: :: Lam     {{ com lambda }}

terminals :: 'terminals_' ::=
| \        :: :: lambda {{ tex \lambda }}
| -->      :: :: red    {{ tex \longrightarrow }}

subrules
        v <:: t

substitutions
        single t x :: tsubst

defns
Jop :: '' ::=

        defn
        t1 --> t2 :: ::reduce::'' {{ com [[t1]] reduces to [[t2]]}} by

        -------------------------- :: ax_app
        (\x.t12) v2 --> {v2/x}t12

        t1 --> t1'
        -------------- :: ctx_app_right
        t t1 --> t t1'

        t1 --> t1'
        -------------- :: ctx_app_left
        t1 v --> t1' v
```

FIGURE 1.9 – λ-calcul avec appel par valeur en Ott.

```
require "modules/substitution.k"

module LAMBDA
        imports SUBSTITUTION

        syntax Val ::= ID
                     | "lambda" ID "." Exp [binder]
        syntax Exp ::= Val
                     | Exp Exp             [left, strict]
                     | "(" Exp ")"         [bracket]
        syntax KResult ::= Val

        rule (lambda X:Id . E:Exp) V:Val => E[V / X]
endmodule
```

FIGURE 1.10 – λ-calcul avec appel par valeur en $\mathbb{K}$.

Langages plongés

Une autre méthode est d'utiliser des langages plongés dans un langage existant, ce qui fournit tous les éléments existant pour le langage hôte, notamment des environnements de développement, mais en garde généralement les limitations. Les langages présentés ci-dessous ne visent pas à être extraits vers d'autres outils.

PLT redex

PLT Redex [2] est un langage dédié pour définir des spécifications de langages de programmation. Il est plongé dans le langage Racket, ce qui fournit un IDE et un ensemble de bibliothèques standard. On trouve en figure 1.11 l'exemple du λ-calcul par valeur écrit avec PLT Redex.

Twelf

Twelf [3] est un langage dédié pour spécifier, implémenter et prouver des propriétés de systèmes déductifs, tels que des langages de programmation, ou des logiques. Il fournit également un mode emacs permettant d'exécuter et de vérifier les codes twelf. L'exemple du λ-calcul avec plongement superficiel (les abstractions sont représentées par des fonctions) est donné en figure 1.12.

Plancomps

PLANCOMPS [7] était un projet de recherche centré sur une approche modulaire des sémantiques. Il factorise les aspects génériques d'une sémantique en utilisant des constructions

2. https ://redex.racket-lang.org/
3. http ://twelf.org

```
#lang racket
(require redex)

(define-language λv
  (e (e e) v)
  (E (v E) (E e) hole)
  (v (λ x e) x)
  (x (variable-except λ)))

(define red
  (reduction-relation
   λv
   (--> (in-hole E ((λ x e) v))
        (in-hole E (subst x v e))
        "ßv")))

(define-metafunction λv
  subst : x any any -> any
  ;; 1. x_1 bound, so don't continue in λ body
  [(subst x_1 any_1 (λ x_1 any_2)) (λ x_1 any_2)]
  ;; 2. general purpose capture avoiding case
  [(subst x_1 any_1 (λ x_2 any_2))
   (λ x_new
     (subst x_1 any_1
            (subst-vars (x_2 x_new) any_2)))
   (where x_new
          ,(variables-not-in
            (term (x_1 any_1))
            (term (x_2 any_2))))]
  ;; 3. replace x_1 with e_1
  [(subst x_1 any_1 x_1) any_1]
  ;; 4. x_1 and x_2 are different, so don't replace
  [(subst x_1 any_1 x_2) x_2]
  ;; the last cases cover all other expressions
  [(subst x_1 any_1 (any_2 any_3))
   ((subst x_1 any_1 any_2) (subst x_1 any_1 any_3))])

(define-metafunction λv
  subst-vars : (x any) any -> any
  [(subst-vars (x_1 any_1) x_1) any_1]
  [(subst-vars (x_1 any_1) (any_2 any_3))
   ((subst-vars (x_1 any_1) any_2) (subst-vars (x_1 any_1) any_3))]
  [(subst-vars (x_1 any_1) any_2) any_2])
```

FIGURE 1.11 – λ-calcul avec appel par valeur en PLT Redex.

```
term : type.    %name term M x.

app : term -> term -> term.
lam : (term -> term) -> term.

%abbrev @ = app.
%infix left 10 @.

step : term -> term -> type.

s-ß : step (app (lam [x] M1 x) (lam F)) (M1 (lam F)).

s-1 : step (app M1 M2) (app M1' M2)
       <- step M1 M1'.

s-2 : step (app M1 M2) (app M1 M2')
       <- step M2 M2'.
```

FIGURE 1.12 – λ-calcul avec appel par valeur en Twelf.

fondamentales. Cela permet de limiter les redondances dans une sémantique, et également de récupérer des morceaux entre plusieurs sémantiques.

Récapitulatif

Le tableau ci-dessous dresse un tableau d'ensemble des avantages et inconvénients de ces différents outils, vis-à-vis de Skel et Necro. Quand nous indiquons qu'un outil ne permet pas une action, il s'agit de ce qui est vrai dans la limite de nos connaissances.

Nous avons choisi plusieurs critères non-exhaustifs, qui nous semblent pertinents pour en juger, il s'agit bien sûr de notre interprétation et de nos usages.

	Lem	Ott	$\mathbb{K}$	PLT Redex	Twelf	Skel / Necro
Exécutable	oui	oui	oui	oui	oui	oui
Extraction vers un assistant de preuve	oui	oui	non	non	non	oui
API pour étendre les outils existants	non	non	non	non	non	oui
Méta-sémantique configurable	non	non	non	non	non	oui
Types natifs	oui	oui	oui	oui	oui	non
Lieurs natifs	non	oui	oui	non	oui	non
Sous-spécification	non	non	non	non	non	oui

Chapitre 2

Skel et les sémantiques squelettiques

Cursed be he that moves my bones.

William Shakespeare

2.1 Skel par l'exemple

Nous décrivons ici comment est spécifiée une sémantique en Skel, en utilisant deux exemples. D'abord la sémantique à petit pas du λ-calcul, sans stratégie d'évaluation, puis la sémantique à grand pas du langage IMP présenté en section 1.1. Ainsi, on a un exemple impératif et un exemple fonctionnel, un exemple petit pas et un exemple grand pas. Une sémantique squelettique est une liste de déclarations de deux sortes différentes. Les *déclarations de types* et les *déclarations de termes*, dont chacun peut être *spécifié* ou *non spécifié*.

Quand un type est non spécifié, on se contente de donner son nom. Par exemple, lorsque l'on définit la sémantique du λ-calcul, on ne souhaitera sans doute pas spécifier comment les variables seront représentés en mémoire interne, et on déclarera donc `type ident`.

Un type spécifié peut-être un *type variant* (c'est-à-dire un type de données algébriques, défini en donnant la liste de ses constructeurs, et de leurs types d'entrée), un *alias de type*, utilisant la notation `:=`, ou un *type enregistrement* (défini en listant ses champs, et les types qu'ils attendent). Par exemple, le type des λ-termes sera un type variant, défini comme suit :

```
type term =
| Var ident
| App (term, term)
| Lam (ident, term)
```

Dans cet exemple, `(ident, term)` est le type produit à deux composantes, dont la première est de type `ident`, et la seconde de type `term`.

Un type enregistrement est déclaré par `type euler_int = (re: int , im: int)` et un alias par `type nat := int`.

Il est à noter que les types sont implicitement mutuellement récursifs, l'ordre choisi pour les définir n'ayant donc aucune importance. En revanche, les alias de types ne sont pas des types à proprement parler, mais uniquement des alias comme leur nom l'indique. Ils peuvent donc être utilisés avec les autres de manière mutuellement récursive, mais ils doivent *in fine* être évaluable vers un type, et donc ne pas contenir de boucle dans leur définition. Le typeur, que nous présenterons en section 3.3, vérifie que les définitions sont non cycliques.

Voyons maintenant les déclarations de termes. La déclaration d'un terme non spécifié est simplement son nom et son type. Pour déclarer un terme spécifié, on donne également sa définition.

Dans notre code, on va par exemple choisir de ne pas spécifier la substitution. On déclare alors `val subst: ident → term → term → term`. Si l'on choisit de donner plus de détail en la spécifiant tout de même, on peut raffiner *a posteriori*. En annexe A.1 par exemple, on a fait le choix de spécifier la substitution.

Un exemple simple de terme spécifié est alors le suivant, qui exécute une étape petit pas dans l'évaluation d'un λ-terme.

```
val ss (t:term): term =
    match t with
    | Var x -> (branch end: term)
    | Lam (x, body) ->
        let b' = ss body in
        Lam (x, b')
    | App (t1, t2) ->
        branch
            let t1' = ss t1 in
            App (t1', t2)
        or
            let t2' = ss t2 in
            App (t1, t2')
        or
            let Lam (x, body) = t1 in
            subst x t2 body (* body[x←t2] *)
        end
    end
```

L'annexe A.1 définit la substitution, et les fonctions utiles pour cette définition.

La reconnaissance de motif avec `match ... with ... end` fonctionne comme on en a l'habitude. Le premier motif qui correspond à l'expression reconnue est sélectionné, et le code qui correspond est exécuté. La construction `branch ... or ... end` est une primitive Skel qui gère les choix non déterministes. Elle est similaire à l'opérateur ambigu de McCarthy [15], dans le sens où un branchement ne donne aucun résultat que dans l'hypothèse où aucune des branches

ne donne un résultat. L'assignation déstructurant `let Lam (x, body) = t1 in` affirme que `t1` est de la forme `Lam (_, _)`. Si cela est vrai, alors les valeurs appropriées seront affectées aux variables x et *body* et l'évaluation continue avec le subst. Sinon, la troisième branche ne s'évaluera à aucun résultat. Le premier branchement est vide, donc `ss (Var x)` ne donnera aucun résultat. En revanche, le deuxième branchement contient plusieurs branches qui peuvent être simultanément valide, et cela fournit du non déterminisme. Il est à noter que contrairement aux branchements, la reconnaissance de motif avec `match ... with` est déterministe.

On constate aussi que le branchement vide indiqué comme squelette pour le cas des variables est explicitement typé. En effet, Le typeur que nous présenterons plus loin ne fait pas d'inférence de type, il faut donc pouvoir décider localement du type d'un branchement, ce qui est impossible avec un branchement vide, si l'on ne le donne pas explicitement.

On constate également que le terme `ss` s'appelle lui-même. De même que pour les types, les termes sont naturellement mutuellement récursifs.

Passons au langage IMP. Le type des expressions et des instructions a été défini formellement en section 1.1 ; on en donne la définition Skel :

```
type expr =
| Const lit
| Var ident
| Plus (expr, expr)
| Equal (expr, expr)
| Not expr

type stmt =
| Skip
| Assign (ident, expr)
| Seq (stmt, stmt)
| If (expr, stmt, stmt)
| While (expr, stmt)
```

Servent de base à cela les types non spécifiés `type ident` et `type lit`.
Contrairement au λ-calcul, les valeurs ne sont pas des termes mais bien des éléments d'un nouvel ensemble. Il faut donc des types pour définir les valeurs. Pour les expressions, on peut avoir des valeurs entières ou booléennes. On garde les entiers non spécifiés, supposant qu'ils peuvent être représentés de bien des manières différentes, suivant les architectures notamment. Il est toujours possible de modifier la sémantique a posteriori pour ajouter la spécification des entiers si l'on juge que leur spécification ne doit pas dépendre de l'implémentation. En revanche, on spécifie les booléens.

```
type value = | Int int | Bool boolean
```

```
type int
type boolean = | True | False
```

Techniquement, comme on le verra dans la section 2.2, tous les constructeurs prennent un argument. Ainsi, la définition ci-dessus pour le type `boolean` est une méta-notation pour la définition suivante :

```
type boolean = | True () | False ()
```

De la même manière dans les termes, on pourra noter indifféremment `True ()` et `True`.

Les instructions sont calculées dans un état donné, et le résultat de leur évaluation est l'état atteint après les avoir évaluées. Ainsi, il nous faut donc un type pour représenter ces états, que l'on choisira de ne pas spécifier. On définit donc :

```
type state
```

On aura aussi besoin de fonctions de bases pour manipuler les types non spécifiés : analyser un littéral, lire une variable dans un état, modifier un état, faire des additions et comparaisons d'égalité sur les entiers :

```
val litToVal : lit → value
val read : ident → state → value
val write : ident → state → value → state
val add : int → int → int
val eq : int → int → boolean
```

On définit également la négation booléenne :

```
val neg (b:boolean): boolean =
    match b with
    | True → False
    | False → True
    end
```

Enfin, on peut définir l'évaluation d'une expression et d'une instruction dans un état donné :

```
val eval_expr (s:state) (e:expr): value =
match e with
| Const i →
    litToVal i
| Var x →
    read x s
| Plus (e1, e2) →
    let Int i1 = eval_expr s e1 in
```

```
    let Int i2 = eval_expr s e2 in
    let i = add i1 i2 in
    Int i
| Equal (e1, e2) →
    let Int i1 = eval_expr s e1 in
    let Int i2 = eval_expr s e2 in
    let b = eq i1 i2 in
    Bool b
| Not e →
    let Bool b = eval_expr s e in
    let b' = neg b in
    Bool b'
end

val eval_stmt (s:state) (t:stmt): state =
    match t with
    | Skip → s
    | Assign (x, e) →
        let v = eval_expr s e in
        write x s v
    | Seq (t1, t2) →
        let s' = eval_stmt s t1 in
        eval_stmt s' t2
    | If (cond, true, false) →
        let Bool b = eval_expr s cond in
        match b with
        | True → eval_stmt s true
        | False → eval_stmt s false
        end
    | While (cond, t') →
        let Bool b = eval_expr s cond in
        match b with
        | True →
            let s' = eval_stmt s t' in
            eval_stmt s' t
        | False → s
        end
    end
```

Cet exemple étant parfaitement déterministe, on n'a pas besoin d'utiliser des branchements.

$$
\begin{array}{rrcl}
\textsc{Term} & t & ::= & x \mid C\ t \mid (t, \ldots, t) \mid \lambda p : \tau \to S \mid t.f \mid t.i \mid (f = t, \ldots, f = t) \mid \\
 & & & t \leftarrow (f = t, \ldots, f = t) \\
\textsc{Skeleton} & S & ::= & t_0\ t_1 \ldots t_n \mid \text{let } p\ =\ S \text{ in } S \mid \text{let } p\ :\ \tau \text{ in } S \mid \\
 & & & \text{match } t \text{ with "}\mid\text{"}p \to S \ldots\text{"}\mid\text{"}p \to S \text{ end} \mid \text{branch } S \text{ or } \ldots \text{ or } S \text{ end} \mid t \\
\textsc{Pattern} & p & ::= & x \mid \_ \mid C\ p \mid (p, \ldots, p) \mid (f = p, \ldots, f = p) \\
\textsc{Type} & \tau & ::= & b \mid \tau \to \tau \mid (\tau, \ldots, \tau) \\
\textsc{Term decl} & d_t & ::= & \text{val } x : \tau \mid \text{val } x : \tau = t \\
\textsc{Type decl} & d_\tau & ::= & \text{type } b \mid \text{type } b = \text{"}\mid\text{"}\ C\ \tau \ldots \text{"}\mid\text{"}\ C\ \tau \mid \text{type } b := \tau \mid \\
 & & & \text{type } b = (f : \tau, \ldots, f : \tau) \\
\textsc{File} & f & ::= & \emptyset \mid d_\tau f \mid d_t f
\end{array}
$$

FIGURE 2.1 – Syntaxe de Skel (sans polymorphisme).

Le code est assez simple, nous détaillons seulement l'exemple du `If` :

Si l'on évalue une instruction conditionnelle, le premier `match` de la fonction `eval_stmt` reconnaîtra le motif `If (cond, true, false)`. On note au passage que `true` et `false` ne sont pas réservés, car Skel ne définit aucun type par défaut. On commence par évaluer la condition, qui doit donner un booléen, et on utilise une reconnaissance de motif déstructurante pour récupérer ce booléen :

```
let Bool b = eval_expr s cond in
```

Puis il ne reste qu'à distinguer suivant `b`, et à évaluer alors une branche du `If` ou la seconde.

2.2 Formalisme

La figure 2.1 définit la syntaxe des termes et des squelettes des sémantiques squelettiques. Elle ne décrit pas le polymorphisme, qui sera présenté dans la Section 2.4. Une BNF complète tenant compte du polymorphisme est proposée en annexe A.2.

Il y a deux types d'expressions, les termes et les squelettes. Héritage de la version présentée dans [5], cette distinction sépare les termes des squelettes. Intuitivement, un terme ne nécessite pas de calcul pour s'évaluer, uniquement des recherches dans l'environnement, et des manipulations simples, tandis qu'un squelette fait des calculs, et peut s'évaluer de manière non déterministe vers une ou plusieurs valeurs, en un temps arbitraire. Cette forme s'approche ainsi du λ-calcul computationnel [16] et de la Forme Normale Administrative [25]. Ce choix assure l'unicité de la stratégie d'évaluation, permet une simplicité de manipulation des expressions, et permet de distinguer explicitement les valeurs des calculs. On en voit notamment les avantages dans la section 4.2, où cette forme permet de définir simplement les monades d'interprétations.

Un terme peut donc être une variable, un constructeur appliqué à un terme, un uplet (possiblement vide) de termes, une λ-abstraction, un enregistrement, l'accès à un membre d'un uplet, l'accès à un champ donné d'un enregistrement, ou la réaffectation de certains champs d'un enregistrement.

Un squelette peut être l'application d'un terme à d'autres termes, une liaison via la construction `let in`, une existentielle (voir section 2.3), un branchement (possiblement vide), ou simplement le renvoi immédiat d'un terme. Dans le dernier cas, on notera parfois $\mathrm{ret}\, t$ au lieu de t, pour insister sur le fait qu'on considère le squelette qui renvoie t et non le terme t.

Un motif est une variable, un motif universel, un constructeur appliqué à un motif, un uplet (possiblement vide) de motifs, ou un motif enregistrement.

Enfin, un type est ou bien un type de base (type défini par l'utilisateurice, spécifié ou non spécifié, ou bien un type flèche, ou enfin un uplet (possiblement vide) de types.

Les déclarations de termes et de types ont quant à elles déjà été décrites dans la section 2.1.

Ces définitions formelles sont agrémentées de quelques sucres syntaxiques pour simplifier l'écriture et la lecture de sémantique. On a notamment les trois sucres syntaxiques suivants :

— Lors de la définition de fonction, on peut indiquer les arguments autrement qu'en utilisant un λ. Au niveau principal du fichier, on pourra noter :

```
term f (x: a): b = skel
```

au lieu de

```
term f: a → b =
    λ x: a → skel
```

Au niveau des liaisons `let in`, on pourra noter :

```
let f (x: a) = s1 in s2
```

au lieu de

```
let f = λ x : a → s1 in
s2
```

— Une deuxième méta-notation, elle aussi plutôt classique, est d'utiliser un `;` pour les `let in` lorsque le squelette lié est ignoré. Ainsi, `sk1 ; sk2` est équivalent syntaxiquement à `let _ = sk1 in sk2`.

— Enfin, on a déjà défini une troisième méta-notation. Lorsqu'un constructeur a un argument de type `()`, celui-ci peut être omis, puisque le type `()` n'a qu'un représentant.

Un projet futur est de restreindre l'application à un seul argument. Cela nous rapprocherait encore de la Forme Normale Administrative, et simplifierait de nombreuses opérations, et de nombreuses applications dans les outils qui manipulent les sémantiques squelettiques. L'inconvénient majeur est que cela rendrait les applications multiples plus lourdes et moins lisibles. On peut éviter ce problème en décurryfiant les fonctions. On s'éloigne ainsi du langage OCaml notamment, qui encourage à la curryfication. À nos yeux, cela rend plus explicite le fait que lors

d'une application multiple, les applications sont en réalité opérées une par une, en appliquant les éventuels effets de bord au fur et à mesure. Aucune décision n'a été prise pour l'instant, et le travail futur consistera à peser les options pour choisir ce qui nous semble le plus pertinent.

En termes mathématiques, une sémantique squelettique est définie de manière similaire comme la donnée d'un uplet composé de l'ensemble des noms de types non spécifiés, l'ensemble des noms de types construits, l'ensemble des noms de types enregistrements, l'ensemble des noms de constructeurs, l'ensemble des noms de champs, l'ensemble des noms de termes non spécifiés, l'ensemble des noms de termes spécifiés, et des fonctions qui définissent les types associés aux champs et constructeurs, et les types et termes associés au déclarations de noms de termes. On notera D_t l'ensemble des déclaration de termes, et D_τ l'ensemble des déclarations de type.

On définit ensuite chaque ensemble de la même manière que dans la BNF présentée en figure 2.1. L'ensemble des types, par exemple, est l'union disjointe des noms de types construits, enregistrements, et non spécifiés, clos par les opérations de flèche et de produit.

Cette définition formelles des sémantiques squelettiques permet de définir les fonctions suivantes :

- ctype qui à chaque constructeur associe le type attendu et le type retourné. Par exemple, pour le λ-calcul, on a ctype(`App`) = (`(term, term)`, `term`).
- ftype qui à chaque constructeur associe le type du champ et le type l'enregistrement auquel il appartient. Par exemple, pour le type `euler_int` présenté plus haut, on a ftype(`im`) = (`int`, `euler_int`).
- fields qui à chaque type d'enregistrement associe l'ensemble de ses champs avec leur type attendu. Par exemple, pour le type `euler_int` présenté plus haut, on a fields(`euler_int`) = {`re` : `int`, `im` : `int`}.
- tspec qui associe son type à chaque nom de terme non spécifié. Pour la substitution du λ-calcul, définie plus haut, on a tspec($subst$) = `(ident, term, term) → term`
- tdef qui associe un type et un terme à chaque nom de terme spécifié. Par exemple, si l'on note S pour le squelette du λ-calcul défini plus haut, on a tdef(ss) = (`term → term`, S)

2.3 Existentielles

Les existentielles sont une construction ajoutée à Skel pour mieux rendre compte du côté non déterministe des sémantiques. Dans certains contextes, on a besoin de décrire des sémantiques en « devinant » des valeurs. C'est le cas par exemple du typage à la Curry, où le type de l'argument d'une λ-abstraction est deviné pour pouvoir typer le λ-terme tout entier. La figure 2.2 montre comment le typage à la Curry est défini par règle d'inférence, et comment cela se traduit en Skel.

Cet exemple met par ailleurs en exergue le fait que Skel ne se limite pas aux sémantiques dynamiques, comme l'évaluation, mais permet également de parler de propriétés statiques, comme le typage.

Une différence entre la version par règles d'inférence et la version squelettique, est que

$$\frac{\Gamma, x : \tau \vdash t : \tau'}{\Gamma \vdash \lambda x \cdot t : \tau \to \tau'}$$

```
val type_term (e: env) (t:term): lambdatype =
    match t with
    | Lam (x, t') →
        let tau : lambdatype in (* existentielle *)
        let e' = extenv e x tau in
        let nu = type_term e' t' in
        Arrow (tau, nu)
    | ...
    end
```

FIGURE 2.2 – Typage à la Curry.

la version squelettique a des entrées et des sorties. Le type du λ-terme est attendu en sortie, et ne peut donc pas être détruit. Ici, la totalité des éléments sont présents en conclusion de la règle d'inférence. On pourrait donc contourner le problème en définissant plutôt `val type_term: env → term → lambdatype → boolean`. On a alors un validateur de bon typage plutôt qu'un calculateur, mais on capturerait alors la règle de typage à la Curry. Cependant, on a d'autres exemples où des éléments nécessaires dans les prémisses ne sont pas présents dans la conclusion. Voici par exemple une règle de sous-typage.

$$\frac{\Gamma \vdash t : \tau' \qquad \tau' \prec \tau}{\Gamma \vdash t : \tau}$$

Ici, le type τ' est « oublié » par la règle d'inférence, et même en utilisant un `type_term` qui valide le bon typage au lien de le calculer, on est obligés de quantifier universellement pour τ'.

2.4 Polymorphisme

Pour des raisons pratiques, y compris la définition de lieurs monadiques (voir section 2.5), le système de types supporte le polymorphisme. Toutes les annotations de types y sont explicites ; elle sont spécifiées en utilisant des chevrons. Tous les types déclarés peuvent être polymorphes.

D'abord les arguments de types des types non spécifiés sont déclarés avec des `_` puisqu'on n'a pas besoin des arguments de type dans la suite.

```
type list<_>
```

Puis, pour les types spécifiés et les types enregistrement, on déclare les arguments de type. Tous les constructeurs et champs de ces types prendront les mêmes arguments de type, on ne le précise donc pas dans la déclaration.

```
(* Variant *)
type union<a,b> =
| InjL a | InjR b
```

```
(* Record *)
type pair<a,b> = (left: a, right: b)
```

Enfin, pour les alias de types, on déclare les arguments de type, et on donne en définition un type quelconque, qui peut utiliser les arguments de type comme des variables de type.

```
type option<a> := union<a,()>
```

On a également la possibilité d'avoir du polymorphisme dans les termes. Voici l'exemple de `map`, qui applique une fonction à chaque élément d'une liste. Les annotations de types y sont données explicitement pour construire un terme (par ex. `Nil<b>`) ou en appelant un terme polymorphe (par ex. `map<a,b> f qa`), mais elles peuvent être inférées localement dans les motifs, et n'y sont donc pas spécifiées (par ex. `let Nil = l in ...`).

```
val map<a,b> (f: a → b) (l: list<a>): list<b> =
branch
    let Nil = l in
    Nil<b>
or
    let Cons (a, qa) = l in
    let b = f a in
    let qb = map<a,b> f qa in
    Cons<b> (b, qb)
end
```

Les termes définis dans le fichier peuvent être polymorphes, mais les termes créés par des liaisons avec `let in` sont nécessairement monomorphes. Cette restriction est due au typage explicite, et pourrait être levée s'il on ajoutait de l'inférence de type. Le typage explicite réduit le risque d'erreur, mais nous pourrions tout de même dans le futur ajouter un système optionnel d'inférence de type. Ce système pourrait également inférer le type des termes définis au niveau supérieur, et les types des arguments dans les λ-abstractions.

2.5 Monades en Skel

Certaines spécifications exécutables de sémantiques, comme ECMA-262 [27] par exemple, utilisent les monades de manière extensive, par exemple pour porter des états, propager des exceptions, ou suspendre des calculs. Skel a un système intégré pour gérer des lieurs monadiques, ce qui rend plus facile de les utiliser, et ce qui rend les codes écrits en Skel plus lisibles et plus proches de la spécification [13].

Supposons qu'un terme `bind<a,b>` est défini, de type `m<a> → (a → m<b>) → m<b>` où `m<·>` est par exemple une monade d'état. Alors, on peut utiliser la notation suivante :

```
let p =%bind s1 in s2
```

Dans ce cas, les arguments de type pour `bind` sont inférés localement (ils peuvent aussi être spécifiés explicitement), et le résultat est sémantiquement équivalent à `bind s1 (λp → s2)`, avec les annotations de type appropriées.

Par exemple, on peut réécrire la sémantique de IMP dans une monade d'état, ce qui rend le code plus lisible, et assure qu'il n'y a pas d'erreur dans les manipulations d'états. Voici la définition de la monade d'état.

```
type st<a> := state → (a, state)
val bind<a, b> (a:st<a>) (f:a → st<b>): st<b> =
    λ s : state → let (a, s) = a s in f a s
val ret<a> (a:a): st<a> = λ s : state → (a, s)
```

En utilisant cette monade, on peut réécrire l'évaluation d'une boucle while, présentée en section 2.1 de la manière suivante.

```
val eval_stmt (t:stmt): st<()> =
match t with
| While (cond, t') →
    let Bool b =%bind eval_expr cond in
    match b with
    | True →
        eval_stmt t' ;%bind
        eval_stmt t
    | False →
        ret<()> ()
    end
| ...
end
```

On a défini plus haut que `s1; s2` est un sucre syntaxique pour `let _ = s1 in s2`. De la même manière, `s1 ;%bind s2` est un sucre syntaxique pour `let _ =%bind s1 in s2`.

Le type de `bind` n'a pas besoin d'être le type d'un lieur monadique pour être utilisé dans la notation précédente. Cela fonctionne avec n'importe quel type, attendu que le second argument de `bind` est un type flèche, et l'interprétation est exactement la même. Il est nécessaire d'avoir cette tolérance quand on travaille avec plusieurs monades à la fois, ou quand un opérateur de liaison n'est pas monadique.

Pour simplifier l'écriture et améliorer la lisibilité, on peut choisir d'affecter des symboles aux lieurs. Par exemple, on peut affecter au lieur de monade d'état le symbole `@s`. On réécrit `eval_stmt` en faisant ce remplacement :

```
binder @s := bind

val eval_stmt (t:stmt): st<()> =
match t with
| While (cond, t') →
    let Bool b =@s eval_expr cond in
    match b with
    | True →
        eval_stmt t' ;@s
        eval_stmt t
    | False →
        ret<()> ()
    end
| ...
end
```

2.6 Typage

Skel est un langage fortement typé avec annotations de type explicites. Chaque terme est déclaré avec son type, ainsi que chaque λ-abstraction. Le polymorphisme, présenté dans la section 2.4, utilise aussi des arguments de types spécifiés. Tout spécifier peut sembler fastidieux au premier abord, mais cela aide à augmenter la confiance dans la correction du code Skel.

Pour donner les règles de typage de Skel, on utilise les fonctions ctype, ftype et fields définies en section 2.2.

Il est à noter qu'un nom de constructeur et un nom de champ ne peuvent être utilisés qu'une unique fois, pour un seul type. Cette unicité est vérifiée par le typeur de Necro Lib, présenté en section 3.3.

Pour donner les règles de typage, il faut ensuite définir les environnements de typage. Un environnement de typage Γ est une liste d'associations des noms de variables vers des types. On définit :

$$\Gamma ::= \epsilon \mid \Gamma, x : \tau$$

On assimilera les environnements de typage à des fonctions partielles des noms de variables vers les types de la manière habituelle. On a donc :

$$(\Gamma, x : \tau)(y) = \begin{cases} \tau & \text{, si } x = y \\ \Gamma(y) & \text{, sinon} \end{cases}$$

Les règles de typage des termes et squelettes sont directes, elles sont données en figure 2.3. Elles supposent qu'on est dans une sémantique donnée, ou ftype, ctype etc. sont déjà définies. Elles

sont respectivement de la forme $\Gamma \vdash t : \tau$ et $\Gamma \vdash S : \tau$, où Γ est un environnement de typage. Ces règles sont globalement similaire à ce qui existe dans d'autres langages fonctionnels.

Nous présentons ici les règles sans tenir compte du polymorphisme. La version polymorphe est sensiblement similaire, elle est présentée en annexe A.2. Le polymorphisme influence uniquement sur le typage. En effet, comme en OCaml par exemple, les types servent à garantir des propriétés de sûreté, mais n'ont pas d'influence sur la sémantique des termes et des squelettes.

Dans les règles de typage des λ-abstractions et des liaisons, on utilise la fonction partielle $\Gamma + p \mapsto \tau$ définie de la manière suivante :

$$\Gamma + x \mapsto \tau \triangleq \Gamma, x : \tau \qquad \Gamma + \_ \mapsto \tau \triangleq \Gamma$$
$$\Gamma + C\ p \mapsto \tau' \triangleq \Gamma + p \mapsto \tau, \text{ où } \operatorname{ctype}(C) = (\tau, \tau')$$
$$\Gamma + (p_1, \ldots, p_n) \mapsto (\tau_1, \ldots, \tau_n) \triangleq ((\Gamma + p_1 \mapsto \tau_1) \ldots) + p_n \mapsto \tau_n$$
$$\Gamma + (f_1 = p_1, \ldots, f_n = p_n) \mapsto \tau' \triangleq ((\Gamma + p_1 \mapsto \tau_1) \ldots) + p_n \mapsto \tau_n,$$
$$\text{où } \operatorname{fields}(\tau') = \{f_1 : \tau_1, \ldots, f_n : \tau_n\}$$

Une fois qu'on a défini comment vérifier le typage d'un terme et d'un squelette, une sémantique squelettique est bien typée lorsque pour chaque déclaration de terme spécifié `val x : τ = t`, on a bien $\epsilon \vdash t : \tau$.

2.7 Interprétation concrète

Skel n'est qu'un langage, une syntaxe concrète pour décrire un langage de programmation. La sémantique associée à une description Skel est appelée une *interprétation*. Nous présentons dans cette section une interprétation concrète, qui correspond à la sémantique naturelle d'un langage [11]. Dans la prochaine section, nous définirons une interprétation abstraite, où les types non spécifiés sont associés à des valeurs dans un domaine abstraite, et où les résultats des branches sont rassemblés.

Chaque squelette et chaque terme est interprété comme une valeur. Il nous faut donc d'abord définir l'ensemble dans lequel nous choisissons ces valeurs.

2.7.1 Ensembles de valeurs

Intuitivement

Tout d'abord, pour tout type non spécifié b, on suppose donné un ensemble de valeurs V_b.

Ensuite, pour tout type variant b, on aura

$$V_b \triangleq \{C\ v \mid \operatorname{ctype}(C) = (\tau, b) \wedge v \in V_\tau\}.$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ \textsc{Var} \qquad \frac{(\texttt{val x:}\tau) \in D_t}{\Gamma \vdash x : \tau}\ \textsc{TermUnspec} \qquad \frac{(\texttt{val x:}\tau\ \texttt{=}\ \texttt{t}) \in D_t}{\Gamma \vdash x : \tau}\ \textsc{TermSpec}$$

$$\frac{\Gamma \vdash t : \tau \qquad \text{ctype(C)} = (\tau, \tau')}{\Gamma \vdash \mathrm{C}\, t : \tau'}\ \textsc{Const} \qquad \frac{\Gamma \vdash t_1 : \tau_1 \qquad \dots \qquad \Gamma \vdash t_n : \tau_n}{\Gamma \vdash (t_1, \dots, t_n) : (\tau_1, \dots, \tau_n)}\ \textsc{Tuple}$$

$$\frac{\Gamma + p \mapsto \tau \vdash S : \tau'}{\Gamma \vdash (\lambda p : \tau \to S) : \tau \to \tau'}\ \textsc{Clos} \qquad \frac{\Gamma \vdash t : \tau' \qquad \text{ftype}(f) = (\tau, \tau')}{\Gamma \vdash t.f : \tau}\ \textsc{FieldGet}$$

$$\frac{\Gamma \vdash t : (\tau_1, \dots, \tau_n) \qquad 1 \leq i \leq n}{\Gamma \vdash t.i : \tau_i}\ \textsc{TupleGet}$$

$$\frac{\text{fields}(\tau) = \{f_1 : \tau_1, \dots, f_n : \tau_n\} \qquad \Gamma \vdash t_1 : \tau_1 \qquad \dots \qquad \Gamma \vdash t_n : \tau_n}{\Gamma \vdash (f_1 = t_1, \dots, f_n = t_n) : \tau}\ \textsc{Rec}$$

$$\frac{\Gamma \vdash t : \tau \qquad \forall i \in [\![1; m]\!], \Gamma \vdash t_i : \tau_i \qquad \forall i \in [\![1; m]\!], \text{ftype}\, f_i = (\tau_i, \tau)}{\Gamma \vdash t \leftarrow (f_1 = t_1, \dots, f_m = t_m) : \tau}\ \textsc{FieldSet}$$

$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash \text{ret}\, t : \tau}\ \textsc{Ret} \qquad \frac{\Gamma \vdash S_1 : \tau \qquad \dots \qquad \Gamma \vdash S_n : \tau}{\Gamma \vdash \text{branch}\ \ S_1\ \text{or}\ \dots\ \text{or}\ \ S_n\ \ \text{end} : \tau}\ \textsc{Branch}$$

$$\frac{\Gamma \vdash S : \tau \qquad \Gamma + p \mapsto \tau \vdash S' : \tau'}{\Gamma \vdash \text{let}\, p = S \,\text{in}\, S' : \tau'}\ \textsc{LetIn} \qquad \frac{\Gamma + p \mapsto \tau \vdash S : \tau'}{\Gamma \vdash \text{let}\, p : \tau \,\text{in}\, S : \tau'}\ \textsc{Exist}$$

$$\frac{\Gamma \vdash t : \tau \qquad \Gamma + p_1 \mapsto \tau \vdash S_1 : \tau' \qquad \dots \qquad \Gamma + p_n \mapsto \tau \vdash S_n : \tau'}{\Gamma \vdash \text{match}\, t\, \text{with} \mid p_1 \to S_1 \mid \dots \mid p_n \to S_n\, \text{end} : \tau'}\ \textsc{Match}$$

$$\frac{\Gamma \vdash t_0 : \tau_1 \to \dots \to \tau_n \to \tau \qquad \Gamma \vdash t_1 : \tau_1 \qquad \dots \qquad \Gamma \vdash t_n : \tau_n}{\Gamma \vdash (t_0\ t_1 \dots t_n) : \tau}\ \textsc{App}$$

FIGURE 2.3 – Règles de typage des sémantiques squelettiques.

Pour les types produits, on définit

$$V_{(\tau_1,\ldots,\tau_n)} \triangleq V_{\tau_1} \times \cdots \times V_{\tau_n}.$$

Pour les types enregistrement, l'ensemble est celui des familles de valeurs avec les types attendus. Par exemple, si on prend un type d'enregistrement b tel que $\text{fields}(b) = \{f_1 : \tau_1, \ldots, f_n : \tau_n\}$, alors V_b est défini par

$$V_b \triangleq \{(f_1 = v_1, \ldots, f_n = v_n) \mid v_i \in V_{\tau_i}\}.$$

Pour les types flèches, on a d'abord envie d'utiliser des relations, puisqu'un squelette peut être évalué vers un nombre arbitraire de valeurs différentes, en raison du non déterminisme et de la partialité (voir plus bas) On voudrait donc définir :

$$V_{\tau\to\tau'} \triangleq \mathcal{R}(V_\tau, V'_\tau) = \mathcal{P}(V_\tau \times V'_\tau).$$

On voit un problème avec le cas d'un type défini par `type a = | A (a → a)`. En effet, on aurait alors $V_a \simeq \mathcal{R}(V_a, V_a)$, ce qui est impossible en raison du théorème de Cantor-Bernstein. Il existe plusieurs moyens pour résoudre ce souci. La manière la plus simple est d'utiliser des ensembles qui permettent de décrire les fonctions de manière opérationnelles. Les fonctions sont donc représentées par des clôtures, lorsqu'il s'agit de λ-abstractions, ou par le nom de la fonction et les arguments déjà donnés lorsqu'il s'agit d'une application partielle de terme non spécifié. Si l'on choisit plus tard de restreindre l'application à un seul terme comme proposé plus haut, on pourrait supprimer le cas des application partielles de termes non spécifiés, puisque l'application partielle se ferait alors via une clôture, qui serait donc gérée par le premier cas.

Pour formaliser cela il faut déjà définir les environnements. On définit les environnements, de manière similaire en tous points aux environnements de typage, à l'exception du fait que les noms de variables sont associés à des valeurs et non à des types :

$$E ::= \epsilon \mid E, x : v$$

On dit que l'environnement E et l'environnement de typage Γ sont cohérents, et on note $\text{OkEnv}(\Gamma \mid E)$, si les noms de variables sont les mêmes et si les types correspondent. Formellement, On définit inductivement $\text{OkEnv}(\cdot \mid \cdot)$ par les deux règles suivantes :

$$\frac{}{\text{OkEnv}(\epsilon \mid \epsilon)} \qquad \frac{v \in V_\tau \qquad \text{OkEnv}(\Gamma \mid E)}{\text{OkEnv}(\Gamma, x : \tau \mid E, x : v)}$$

Maintenant, on définit la fonction closures qui rassemble toutes les clôtures utilisées dans une

sémantique squelettique Σ :

$$\text{closures}^{(\alpha,\beta)}(\Sigma) = \bigcup_{(\texttt{term x:}\tau\ \texttt{=}\ \texttt{t})\in D_t(\Sigma)} \text{closures}_t^{(\alpha,\beta)}(t,\tau,\epsilon).$$

$\text{closures}_t^{(\alpha,\beta)}$ et $\text{closures}_S^{(\alpha,\beta)}$ sont définis récursivement sur l'ensemble des termes et des squelettes de la manière suivante :

- $\text{closures}_t^{(\alpha,\beta)}(x,\_,\_) = \emptyset$.
- $\text{closures}_t^{(\alpha,\beta)}(C\ t,\tau,\Gamma) = \text{closures}_t^{(\alpha,\beta)}(t,\tau',\Gamma)$ où $\text{ctype}(C) = (\tau',\tau)$.
- $\text{closures}_t^{(\alpha,\beta)}((t_1,\ldots,t_n),(\tau_1,\ldots,\tau_n),\Gamma) = \bigcup_{1\leq i\leq n} \text{closures}_t^{(\alpha,\beta)}(t_i,\tau_i,\Gamma)$.
- si $(\tau,\tau') \neq (\alpha,\beta)$, alors $\text{closures}_t^{(\alpha,\beta)}(\lambda p^\tau \cdot S, \tau\to\tau',\Gamma) = \text{closures}_S^{(\alpha,\beta)}(S,\tau',\Gamma + p\mapsto\tau)$.
- $\text{closures}_t^{(\alpha,\beta)}(\lambda p:\alpha\to S,\alpha\to\beta,\Gamma) = \{(p,S,\Gamma)\} \cup \text{closures}_S^{(\alpha,\beta)}(S,\beta,\Gamma + p\mapsto\alpha)$.
- $\text{closures}_t^{(\alpha,\beta)}(t.f,\tau,\Gamma) = \text{closures}_t^{(\alpha,\beta)}(t,\tau',\Gamma)$ où $\text{ftype}(f) = (\tau,\tau')$.
- $\text{closures}_t^{(\alpha,\beta)}(t.i,(\tau_1,\ldots,\tau_n),\Gamma) = \text{closures}_t^{(\alpha,\beta)}(t,\tau_i,\Gamma)$ où $1\leq i\leq n$.
- $\text{closures}_t^{(\alpha,\beta)}((f_1=t_1,\ldots,f_n=t_n),\tau,\Gamma) = \bigcup_{1\leq i\leq n} \text{closures}_t^{(\alpha,\beta)}(t_i,\tau_i,\Gamma)$,
  où $\text{fields}(\tau) = \{f_1:\tau_1,\ldots,f_n:\tau_n\}$.
- Lorsque $\{f_1:\tau_1,\ldots,f_m:\tau_m\} \subseteq \text{fields}(\tau)$,

$$\text{closures}_t^{(\alpha,\beta)}(t \leftarrow (f_1=t_1,\ldots,f_m=t_m),\tau,\Gamma) = \text{closures}_t^{(\alpha,\beta)}(t,\tau,\Gamma) \cup \bigcup_{1\leq i\leq m} \text{closures}_t^{(\alpha,\beta)}(t_i,\tau_i,\Gamma).$$

- Lorsque $\forall i \in [\![1;n]\!].\ \Gamma \vdash t_i : \tau_i$,

$$\text{closures}_S^{(\alpha,\beta)}(f\ t_1\ \ldots\ t_n,\tau,\Gamma) = \text{closures}_t^{(\alpha,\beta)}(f,\tau_1\to\cdots\to\tau_n\to\tau,\Gamma) \cup \bigcup_{1\leq i\leq n} \text{closures}_t^{(\alpha,\beta)}(t_i,\tau_i,\Gamma).$$

- Lorsque $\Gamma \vdash S_1 : \tau'$,

$$\text{closures}_S^{(\alpha,\beta)}(\text{let}\, p = S_1 \,\text{in}\, S_2,\tau,\Gamma) = \text{closures}_S^{(\alpha,\beta)}(S_1,\tau',\Gamma) \cup \text{closures}_S^{(\alpha,\beta)}(S_2,\tau,\Gamma + p\mapsto\tau').$$

- $\text{closures}_S^{(\alpha,\beta)}(\text{let}\, p:\tau' \,\text{in}\, S,\tau,\Gamma) = \text{closures}_S^{(\alpha,\beta)}(S,\tau,\Gamma + p\mapsto\tau')$.
- Lorsque $\Gamma \vdash t : \tau'$,

$$\text{closures}_S^{(\alpha,\beta)}(\text{match}\, t \,\text{with} \mid p_1\to S_1 \mid \ldots \mid p_n\to S_n \,\text{end},\tau,\Gamma) = \text{closures}_t^{(\alpha,\beta)}(t,\tau',\Gamma) \cup \bigcup_{1\leq i\leq n} \text{closures}_S^{(\alpha,\beta)}(S_i,\tau,\Gamma + p_i\mapsto\tau').$$

- $\text{closures}_S^{(\alpha,\beta)}(\text{branch}\, S_1 \,\text{or}\ldots\text{or}\, S_n \,\text{end},\tau,\Gamma) = \bigcup_{1\leq i\leq n} \text{closures}_S^{(\alpha,\beta)}(S_i,\tau,\Gamma)$.

— $\text{closures}_S^{(\alpha,\beta)}(\text{ret}\, t, \tau, \Gamma) = \text{closures}_t^{(\alpha,\beta)}(t, \tau, \Gamma)$.

On définit finalement $\text{Clos}(\tau, \tau') \triangleq \{\langle p, E, S\rangle \mid \exists \Gamma, \text{OkEnv}(\Gamma \mid E) \wedge (p, S, \Gamma) \in \text{closures}^{(\tau,\tau')}(\Sigma)\}$

On gère ensuite le cas des termes non spécifiés. On note $\lceil x, \texttt{args}\rceil$ pour représenter le terme x appliqué partiellement aux arguments `args`. On définit ensuite :

$$\text{Unspec}(\tau) \triangleq \{\lceil x, (v_1, \ldots, v_m)\rceil \mid m \in \mathbb{N}, \text{tspec}(x) = \alpha_1 \to \cdots \to \alpha_m \to \tau, v_1 \in V_{\alpha_1}, \ldots, v_m \in V_{\alpha_m}\}$$

On a alors :

$$V_{\tau\to\tau'} = \text{Clos}(\tau, \tau') \cup \text{Unspec}(\tau \to \tau').$$

Formellement

Si l'on considère le type `nat` défini par `type nat = | O | S nat`, on aurait alors une définition circulaire :

$$V_{\texttt{nat}} \triangleq \{\texttt{O}\} \cup \{\texttt{S}\ i \mid i \in V_{\texttt{nat}}\}.$$

Si l'on comprend bien l'idée derrière cette définition, on voit néanmoins que ce n'est pas acceptable formellement. On définit donc ces ensembles de manière inductive. On définit pour chaque type τ et pour tout entier n un ensemble de valeur $V_\tau^{(n)}$. On définit ces ensembles par récurrence sur n, τ muni de l'ordre lexicographique $\prec_{lex}$, où l'ordre $\prec$ sur les types est l'ordre inductif $(\tau_i \prec (\tau_1, \ldots, \tau_n), i = 1, \ldots, n$, et $\tau_i \prec \tau_1 \to \tau_2, i = 1, 2)$.

(non spécifié) Pour tout type τ non spécifié, on suppose donné une fonctionnelle F croissante, et on pose $V_\tau^{(n+1)} \triangleq F\left((V_\tau^{(n)})_{\tau\in S}\right)$.

(variant) Pour un type variant b, on pose $V_b^{(0)} \triangleq \emptyset$ et

$$V_b^{(n+1)} \triangleq V_b^{(n)} \cup \{C\ v \mid \text{ctype}(C) = (\tau, b) \wedge v \in V_\tau^{(n)}\}.$$

Cette définition est acceptable car $(n, \tau) \prec_{lex} (n + 1, b)$.

(enregistrement) Si b est un type d'enregistrement tel que $\text{fields}(b) = \{f_1 : \tau_1, \ldots, f_n : \tau_n\}$, alors on a $V_b^{(0)} = \emptyset$ et

$$V_b^{(m+1)} \triangleq V_b^{(m)} \cup \{(f_1 = v_1, \ldots, f_n = v_n) \mid v_i \in V_{\tau_i}^{(m)}\}.$$

Cette définition est acceptable pour les mêmes raisons que plus haut.

(produit) On pose $V^{(m)}_{(\tau_1,\ldots,\tau_n)} = V_{\tau_1}^{(m)} \times \cdots \times V_{\tau_n}^{(m)}$. Cette définition est acceptable par la définition de $\prec$.

(flèche) On pose enfin $V^{(0)}_{\tau\to\tau'} = \emptyset$ et

$$V^{(n+1)}_{\tau\to\tau'} = \text{Clos}^{(n)}(\tau \to \tau') \cup \text{Unspec}^{(n)}(\tau \to \tau').$$

$\text{Clos}^{(n)}$ est défini comme Clos plus haut, mais avec un environnement contenant des valeurs dans les $V_\alpha^{(n)}$, et il en est de même pour $\text{Unspec}^{(n)}$ Cette définition est acceptable car

$(n, \alpha) \prec_{lex} (n+1, \tau \to \tau')$.

On vérifie que pour tout (n, τ), on a $V_\tau^{(n)} \subseteq V_\tau^{(n+1)}$. On pose enfin pour tout τ :

$$V_\tau \triangleq \bigcup_{i=0}^{+\infty} V_\tau^{(i)}.$$

Alors, on a bien les 4 égalités intuitives, à savoir :

— Pour b variant,

$$V_b = \{C\ v \mid \text{ctype}(C) = (\tau, b) \wedge v \in V_\tau\}.$$

— Pour les types produits,

$$V_{(\tau_1,\ldots,\tau_n)} = V_{\tau_1} \times \cdots \times V_{\tau_n}.$$

— Pour un type d'enregistrement b tel que $\text{fields}(b) = \{f_1 : \tau_1, \ldots, f_n : \tau_n\}$,

$$V_b = \{(f_1 = v_1, \ldots, f_n = v_n) \mid v_i \in V_{\tau_i}\}.$$

— Enfin, pour les types flèches,

$$V_{\tau \to \tau'} = \text{Clos}(\tau, \tau') \cup \text{Unspec}(\tau \to \tau').$$

Les 4 preuves se font par doubles inclusions classiques et sont laissées en exercice.

2.7.2 Règles d'inférence (version inductive)

Ensuite, pour pouvoir donner du sens à tous les termes et squelette, il faut choisir comment on donne du sens à des termes non spécifiés. Avant toute chose, on définit les fonctions d'extensions d'environnement.

On définit dans la figure 2.4 la relation $E + p \mapsto v = E'$ qui affirme que l'environnement E peut être étendu en E' en reconnaissant v contre p, la relation $p \not\mapsto v$ qui affirme que v ne correspond pas au motif p, et la relation $E + v \,\text{match}\, p_1 \ldots p_n = (i, E')$ qui affirme que lors de la reconnaissance de motif de v contre les motifs $p_1, \ldots, p_n$, le premier motif qui correspond est le motif p_i, et on a $E + p_i \mapsto v = E'$.

On remarque notamment que pour tout environnement E, pour tout motif p de type τ, et pour toute valeur v de type τ, on a ou bien $p \not\mapsto v$, ou alors il existe un unique E' tel que $E + p \mapsto v = E'$.

Ensuite, on définit comment interpréter les termes et squelettes. Pour cela, on suppose donnés pour chaque terme non spécifié `val x` : τ un entier m noté $\text{arity}(x)$ tel que $\tau = \alpha_1 \to \cdots \to \alpha_m \to \tau'$ [1] et, si $m \neq 0$, une relation (m+1)-aire $(\!|x|\!) \in \mathcal{P}(V_{\alpha_1} \times \cdots \times V_{\alpha_m} \times V_{\tau'})$. Dans le cas où $m = 0$, on demande $(\!|x|\!) \in V'_\tau$.

1. Le type τ' peut lui-même être un type flèche, si l'arité est choisie non maximale.

$$\frac{}{E + x \mapsto v = E, x : v}\ \textsc{ExtVar} \qquad \frac{}{E + \_ \mapsto v = E}\ \textsc{ExtWild}$$

$$\frac{E + p \mapsto v = E'}{E + Cp \mapsto Cv = E'}\ \textsc{ExtConstr}$$

$$\frac{E_0 + p_1 \mapsto v_1 = E_1 \qquad \dots \qquad E_{n-1} + p_n \mapsto v_n = E_n}{E_0 + (p_1, \dots, p_n) \mapsto (v_1, \dots, v_n) = E_n}\ \textsc{ExtTuple}$$

$$\frac{E_0 + p_1 \mapsto v_1 = E_1 \qquad \dots \qquad E_{n-1} + p_m \mapsto v_m = E_m \qquad m \leq n}{E_0 + (f_1 = p_1, \dots, f_m = p_m) \mapsto (f_1 = v_1, \dots, f_n = v_n) = E_m}\ \textsc{ExtRecord}$$

$$\frac{p \not\rightarrow v}{Cp \not\rightarrow Cv}\ \textsc{NoMatchSameConstr} \qquad \frac{C \neq C'}{Cp \not\rightarrow C'v}\ \textsc{NoMatchDiffConstr}$$

$$\frac{p_i \not\rightarrow v_i \qquad 1 \leq i \leq n}{(p_1, \dots, p_n) \not\rightarrow (v_1, \dots, v_n)}\ \textsc{NoMatchTuple}$$

$$\frac{p_i \not\rightarrow v_i \qquad 1 \leq i \leq m \leq n}{(f_1 = p_1, \dots, f_m = p_m) \not\rightarrow (f_1 = v_1, \dots, f_n = v_n)}\ \textsc{NoMatchRecord}$$

$$\frac{E + p_1 \mapsto v = E'}{E + v\,\mathrm{match}(p_1, \dots, p_n) = (1, E')}\ \textsc{GetMatchFirst}$$

$$\frac{p_1 \not\rightarrow v \qquad E + v\,\mathrm{match}(p_2, \dots, p_n) = (i, E')}{E + v\,\mathrm{match}(p_1, \dots, p_n) = (i + 1, E')}\ \textsc{GetMatchNext}$$

FIGURE 2.4 – Règles pour manipuler les environnements.

Les règles définissant l'interprétation concrète sont données dans la figure 2.5. Elles sont de la forme $t@E = v$ pour les termes, et $E, S \Downarrow v$ pour les squelettes, ce qui signifie que dans l'environnement E, le terme t ou le squelette S peut s'évaluer à la valeur v.

On définit en parallèle la relation $f\ v_1\ \ldots\ v_m \Downarrow_{\text{app}} w$ qui affirme qu'en appliquant la valeur f aux arguments successifs $v_1, \ldots, v_m$, on peut obtenir w.

L'évaluation est déterministe pour les termes, même si tous les termes n'ont pas forcément une interprétation. Un exemple de terme qui ne possède pas d'interprétation concrète est `term x: () = x`. L'évaluation des squelettes, quant à elle, est relationnelle, puisqu'un squelette peut avoir plusieurs interprétation différentes, notamment en raison des branchements. C'est pour refléter le déterminisme des termes qu'on a choisi d'avoir un $(\!|x|\!)$ déterministe dans le cas d'une arité de 0. La distinction s'exprime naturellement lorsqu'on décurryfie la relation $(m+1)$-aire exprimée plus haut, comme on le verra dans le chapitre 5, puisque les flèches sont traduites dans la catégorie de Kleisli et font apparaître l'aspect relationnel, dès lors que le terme non spécifié est fonctionnel.

Comme mentionné plus haut, l'existentielle est une construction spécifique à Skel. Pour interpréter l'existentielle `let p:`τ `in sk`, on prend n'importe quelle valeur de type τ, on l'associe au motif `p`, puis on évalue la continuation dans l'environnement étendu.

L'interprétation concrète est relationnelle. Un squelette peut être interprété comme 0, 1 ou plusieurs valeurs. Les sources de partialité et de non déterminisme sont les suivantes :

— Un branchement avec aucune branche peut n'avoir aucun résultat et causer de la partialité. Un branchement avec plusieurs branches peut avoir plusieurs résultats et causer du non déterminisme.
— L'assignation déstructurante peut échouer, et est ainsi une source de partialité.
— Les termes non spécifiés, lorsqu'ils sont de type flèches, peuvent définir des fonctions partielles ou non déterministes, qui se révèleront lors de leur application à des arguments.
— Enfin, comme d'habitude, la non-terminaison peut-être une source de partialité.

L'interprétation concrète dans sa version inductive est implémentée en Coq sur le dépôt de Necro Coq. Nous en parlerons en section 5.5.

2.7.3 Règles d'inférence (version itérative)

On donne une version où l'induction est exprimée de manière explicite, en utilisant des ensembles $\mathcal{T}$, $\mathcal{S}$ et $\mathcal{A}$ que l'on met à jour. On assimilera ces ensembles à des relations. Ainsi, la notation $\mathcal{T}(t, E, v)$ est utilisée pour signifier que $(t, E, v) \in \mathcal{T}$.

On doit définir comment faire une étape d'évaluation. Les règles pour cette unique étape sont définies dans la figure 2.6. Elles sont de la forme $[\![t]\!](E, \mathcal{T}) \Downarrow^1 v$ pour les termes, $[\![S]\!](E, \mathcal{T}, \mathcal{S}, \mathcal{A}) \Downarrow^1 v$ pour les squelettes et $[\![v_1\ \ldots\ v_n]\!](\mathcal{T}, \mathcal{S}, \mathcal{A}) \Downarrow^1_{\text{app}} v$ pour les applications. Nous avons choisi une présentation sous forme de règles d'inférences, mais contrairement à la version inductive, ces règles ne sont pas récursives. C'est-à-dire que les prémisses des règles ne font jamais appel à $\Downarrow^1$.

$$\frac{E(x) = v}{x@E = v}\ \textsc{Var} \qquad \frac{t@E = v}{(\mathrm{C}\,t)@E = \mathrm{C}\,v}\ \textsc{Const} \qquad \frac{(\texttt{val x:}\tau\ \texttt{=}\ \texttt{t}) \in D \qquad t@\epsilon = v}{x@E = v}\ \textsc{TermSpec}$$

$$\frac{(\texttt{val x} : \tau) \in D \qquad \mathrm{arity}(x) = 0}{x@E = (\!|x|\!)}\ \textsc{TermUnspecZero} \qquad \frac{(\texttt{val x} : \tau) \in D \qquad \mathrm{arity}(x) > 0}{x@E = \lceil x, () \rceil}\ \textsc{TermUnspecSucc}$$

$$\frac{t_1@E = v_1 \qquad \dots \qquad t_n@E = v_n}{(t_1, \dots, t_n)@E = (v_1, \dots, v_n)}\ \textsc{Tuple}$$

$$\frac{(p, S, \Gamma) \in \mathrm{closures}^{(\tau,\tau')}(\Sigma) \qquad \mathrm{OkEnv}(\Gamma \mid E)}{(\lambda p : \tau \to S)@E = \langle p, E, S \rangle}\ \textsc{Clos}$$

$$\frac{t@E = (f_1 = v_1, \dots, f_n = v_n)}{t.f_i@E = v_i}\ \textsc{FieldGet} \qquad \frac{t@E = (v_1, \dots, v_n) \qquad 1 \le i \le n}{t.i@E = v_i}\ \textsc{TupleGet}$$

$$\frac{t_1@E = v_1 \qquad \dots \qquad t_n@E = v_n}{(f_1 = t_1, \dots, f_n = t_n)@E = (f_1 = v_i, \dots, f_n = v_n)}\ \textsc{Rec}$$

$$\frac{\begin{array}{c} t@E = (f_1 = v_1, \dots, f_n = v_n) \qquad \forall i \in [\![1; m]\!], t_i@E = w_{j_i} \\ \forall i \in [\![1; n]\!] \setminus \{j_1, \dots, j_m\}, w_i = v_i \end{array}}{t \leftarrow (f_{j_1} = t_1, \dots, f_{j_m} = t_m)@E = (f_1 = w_1, \dots, f_n = w_n)}\ \textsc{FieldSet} \qquad \frac{t@E = v}{E, \mathrm{ret}\,t \Downarrow v}\ \textsc{Ret}$$

$$\frac{E, S_i \Downarrow v \qquad 1 \le i \le n}{E, \mathrm{branch}\ \ S_1\ \ \mathrm{or}\ \ \dots\ \ \mathrm{or}\ \ S_n\ \ \mathrm{end} \Downarrow v}\ \textsc{Branch} \qquad \frac{E, S \Downarrow v \qquad E + p \mapsto v, S' \Downarrow w}{E, \mathrm{let}\,p = S\,\mathrm{in}\,S' \Downarrow w}\ \textsc{LetIn}$$

$$\frac{v \in V_\tau \qquad E + p \mapsto v, S \Downarrow w}{E, \mathrm{let}\,p : \tau\,\mathrm{in}\,S \Downarrow w}\ \textsc{Exist}$$

$$\frac{t@E = v \qquad E + v\,\mathrm{match}\,p_1 \dots p_n = (i, E') \qquad E', S_i \Downarrow w}{E, \mathrm{match}\,t\,\mathrm{with} \mid p_1 \to S_1 \mid \dots \mid p_n \to S_n\,\mathrm{end} \Downarrow w}\ \textsc{Match}$$

$$\frac{t_0@E = f \qquad t_1@E = v_1 \qquad \dots \qquad t_n@E = v_n \qquad f\ v_1\ \dots\ v_n \Downarrow_{\mathrm{app}} w}{E, (t_0\ t_1 \dots t_n) \Downarrow w}\ \textsc{App}$$

$$\frac{}{v \Downarrow_{\mathrm{app}} v}\ \textsc{AppZero} \qquad \frac{E + p \mapsto v_1, S \Downarrow g \qquad g\ v_2\ \dots\ v_n \Downarrow_{\mathrm{app}} w}{\langle p, E, S \rangle\ \ v_1\ \dots\ v_n \Downarrow_{\mathrm{app}} w}\ \textsc{AppClos}$$

$$\frac{\mathrm{arity}(x) > n}{\lceil x, (v_1, \dots, v_m) \rceil\ \ v_{m+1}\ \dots\ v_n \Downarrow_{\mathrm{app}} \lceil x, (v_1, \dots, v_n) \rceil}\ \textsc{AppUnspecNext}$$

$$\frac{k < \mathrm{arity}(x) = m \le n \qquad (\!|x|\!)(v_1, \dots, v_m, g) \qquad g\ v_{m+1}\ \dots\ v_n \Downarrow_{\mathrm{app}} w}{\lceil x, (v_1, \dots, v_k) \rceil\ \ v_{k+1}\ \dots\ v_n \Downarrow_{\mathrm{app}} w}\ \textsc{AppUnspecCont}$$

FIGURE 2.5 – Interprétation concrète des sémantiques squelettiques.

$$\frac{E(x) = v}{[\![x]\!](E, \mathcal{T}) \Downarrow^1 v}\ \textsc{Var} \qquad \frac{\texttt{val x : } \tau \qquad \text{arity}(x) = 0}{[\![x]\!](E, \mathcal{T}) \Downarrow^1 (\!|x|\!)}\ \textsc{TermUnspecZero}$$

$$\frac{\texttt{val x : } \tau \qquad \text{arity}(x) > 0}{[\![x]\!](E, \mathcal{T}) \Downarrow^1 \lceil x, () \rceil}\ \textsc{TermUnspecSucc} \qquad \frac{\texttt{val x:}\tau\texttt{ = t} \qquad \mathcal{T}(t, \epsilon, v)}{[\![x]\!](E, \mathcal{T}) \Downarrow^1 v}\ \textsc{TermSpec}$$

$$\frac{\mathcal{T}(t, E, v)}{[\![(\mathrm{C}\, t)]\!](E, \mathcal{T}) \Downarrow^1 \mathrm{C}\, v}\ \textsc{Const} \qquad \frac{\mathcal{T}(t_1, E, v_1) \qquad \dots \qquad \mathcal{T}(t_n, E, v_n)}{[\![(t_1, \dots, t_n)]\!](E, \mathcal{T}) \Downarrow^1 (v_1, \dots, v_n)}\ \textsc{Tuple}$$

$$\frac{(p, S, \Gamma) \in \text{closures}^{(\tau,\tau')}(\Sigma) \qquad \text{OkEnv}(\Gamma \mid E)}{[\![(\lambda p : \tau \to S)]\!](E, \mathcal{T}) \Downarrow^1 \langle p, E, S \rangle}\ \textsc{Clos}$$

$$\frac{\mathcal{T}(t, E, (f_1 = v_1, \dots, f_n = v_n))}{[\![t.f_i]\!](E, \mathcal{T}) \Downarrow^1 v_i}\ \textsc{FieldGet} \qquad \frac{\mathcal{T}(t, E, (v_1, \dots, v_n)) \qquad 1 \le i \le n}{[\![t.i]\!](E, \mathcal{T}) \Downarrow^1 v_i}\ \textsc{TupleGet}$$

$$\frac{\mathcal{T}(t_1, E, v_1) \qquad \dots \qquad \mathcal{T}(t_n, E, v_n)}{[\![(f_1 = t_1, \dots, f_n = t_n)]\!](E, \mathcal{T}) \Downarrow^1 (f_1 = v_i, \dots, f_n = v_n)}\ \textsc{Rec}$$

$$\frac{\begin{array}{c}\mathcal{T}(t, E, (f_1 = v_1, \dots, f_n = v_n)) \qquad \forall i \in [\![1; m]\!], \mathcal{T}(t_i, E, w_{j_i}) \\ \forall i \in [\![1; n]\!] \setminus \{j_1, \dots, j_m\}, w_i = v_i\end{array}}{[\![t \leftarrow (f_{j_1} = t_1, \dots, f_{j_m} = t_m)]\!](E, \mathcal{T}) \Downarrow^1 (f_1 = w_1, \dots, f_n = w_n)}\ \textsc{FieldSet}$$

$$\frac{\mathcal{T}(t, E, v)}{[\![\text{ret}\, t]\!](E, \mathcal{T}, \mathcal{S}, \mathcal{A}) \Downarrow^1 v}\ \textsc{Ret} \qquad \frac{\mathcal{S}(S_i, E, v) \qquad 1 \le i \le n}{[\![\text{branch } S_1 \text{ or } \dots \text{ or } S_n \text{ end}]\!](E, \mathcal{T}, \mathcal{S}, \mathcal{A}) \Downarrow^1 v}\ \textsc{Branch}$$

$$\frac{\mathcal{S}(S_1, E, v) \qquad \mathcal{S}(S_2, E + p \mapsto v, w)}{[\![\text{let } p = S_1 \text{ in } S_2]\!](E, \mathcal{T}, \mathcal{S}, \mathcal{A}) \Downarrow^1 w}\ \textsc{LetIn} \qquad \frac{v \in V_\tau \qquad \mathcal{S}(S, E + p \mapsto v, w)}{[\![\text{let } p : \tau \text{ in } S]\!](E, \mathcal{T}, \mathcal{S}, \mathcal{A}) \Downarrow^1 w}\ \textsc{Exist}$$

$$\frac{\mathcal{T}(t, E, v) \qquad E + v \,\text{match}\, p_1 \dots p_n = (i, E') \qquad \mathcal{S}(S_i, E', w)}{[\![\text{match } t \text{ with} \mid p_1 \to S_1 \mid \dots \mid p_n \to S_n \text{ end}]\!](E, \mathcal{T}, \mathcal{S}, \mathcal{A}) \Downarrow^1 w}\ \textsc{Match}$$

$$\frac{\mathcal{T}(t_0, E, f) \qquad \mathcal{T}(t_1, E, v_1) \qquad \dots \qquad \mathcal{T}(t_n, E, v_n) \qquad \mathcal{A}(f\ v_1\ \dots\ v_n, w)}{[\![(t_0\ t_1 \dots t_n)]\!](E, \mathcal{T}, \mathcal{S}, \mathcal{A}) \Downarrow^1 w}\ \textsc{App}$$

$$\frac{}{[\![v]\!](\mathcal{T}, \mathcal{S}, \mathcal{A}) \Downarrow^1_{\text{app}} v}\ \textsc{AppZero} \qquad \frac{\mathcal{S}(S, E + p \mapsto v_1, g) \qquad \mathcal{A}(g\ v_2\ \dots\ v_n, w)}{[\![\langle p, E, S \rangle\ v_1\ \dots\ v_n]\!](\mathcal{T}, \mathcal{S}, \mathcal{A}) \Downarrow^1_{\text{app}} w}\ \textsc{AppClos}$$

$$\frac{\text{arity}(x) > n}{[\![\lceil x, (v_1, \dots, v_m) \rceil\ v_{m+1}\ \dots\ v_n]\!](\mathcal{T}, \mathcal{S}, \mathcal{A}) \Downarrow^1_{\text{app}} \lceil x, (v_1, \dots, v_n) \rceil}\ \textsc{AppUnspecNext}$$

$$\frac{k < \text{arity}(x) = m \le n \qquad (\!|x|\!)(v_1, \dots, v_m, g) \qquad \mathcal{A}(g\ v_{m+1}\ \dots\ v_n, w)}{[\![\lceil x, (v_1, \dots, v_k) \rceil\ v_{k+1}\ \dots\ v_n]\!](\mathcal{T}, \mathcal{S}, \mathcal{A}) \Downarrow^1_{\text{app}} w}\ \textsc{AppUnspecCont}$$

FIGURE 2.6 – Étape d'évaluation.

On a ensuite

$$\begin{aligned}
\mathcal{H}_t(T) &= \left\{(t, E, v) \mid [\![t]\!](E, \mathcal{T}) \Downarrow^1 v\right\}. \\
\mathcal{H}_S(T, S, A) &= \left\{(S, E, v) \mid [\![S]\!](E, \mathcal{T}, \mathcal{S}, \mathcal{A}) \Downarrow^1 v\right\}. \\
\mathcal{H}_{\text{app}}(T, S, A) &= \left\{((v_1, \ldots, v_n), v) \mid [\![v_1 \ \ldots \ v_n]\!](\mathcal{T}, \mathcal{S}, \mathcal{A}) \Downarrow^1_{\text{app}} v\right\}. \\
\mathcal{H}(T, S, A) &= (\mathcal{H}_t(T), \mathcal{H}_S(T, S, A), \mathcal{H}_{\text{app}}(T, S, A)).
\end{aligned}$$

Et enfin, on note :

$$(\mathbb{T}, \mathbb{S}, \mathbb{A}) = \bigcup_{n \in \mathbb{N}} \mathcal{H}^n(\emptyset, \emptyset, \emptyset).$$

Cette version est plus proche de la version proposée à POPL, et elle permet de traiter également le cas des sémantiques abstraites, présentées plus bas. L'interprétation concrète dans sa version itérative est également implémentée en Coq sur le dépôt de Necro Coq. Nous en parlerons en section 5.5. Sur le dépôt est également présente la preuve de l'équivalence des deux versions. Cette équivalence est le théorème suivant :

Theorem 3.

$$\begin{aligned}
\forall tEv.\ t@E = v &\Leftrightarrow \mathbb{T}(t, E, v) \wedge \\
\forall SEv.\ E, S \Downarrow v &\Leftrightarrow \mathbb{S}(S, E, v) \wedge \\
\forall v_1 \ldots v_n v.\ v_1 \ \ldots \ v_n \Downarrow_{\text{app}} v &\Leftrightarrow \mathbb{A}((v_1, \ldots, v_n), v).
\end{aligned}$$

Ce résultat est prouvé en Coq, et est accessible dans le dépôt de Necro Coq [2].

2.7.4 Subject Reduction et progrès

La subject reduction, ou propriété de préservation du type, est la propriété qui affirme que l'évaluation d'une expression conserve son type. Exprimé en petit pas, il s'agit de dire que si une expression a un type, alors, l'évaluation d'un pas sur cette expression donne une nouvelle expression du même type. La version petit pas de l'interprétation concrète n'est pas présentée ici, mais elle est implémentée en Coq, et présentée dans la section 5.5.

La propriété de subject reduction est vérifiée par Skel, et est prouvée en Coq. La preuve est disponible sur le dépôt de Necro Coq [3].

En grand pas, la propriété de subject reduction implique que l'évaluation d'une expression de type τ donnera une valeur de type τ. Cette propriété est donc également vérifiée par Skel. Plus formellement :

2. https://gitlab.inria.fr/skeletons/necro-coq/-/blob/master/proofs/eq_rec/EqConcrete.v#L453
3. https://gitlab.inria.fr/skeletons/necro-coq/-/blob/master/proofs/typecheck/SubjectReduction.v#L1133

$$\frac{E, S_1 \Downarrow \bot \qquad \ldots \qquad E, S_n \Downarrow \bot}{E, \text{branch } S_1 \text{ or } \ldots \text{ or } S_n \text{ end} \Downarrow \bot}\ \textsc{Branch} \qquad \frac{E, S \Downarrow v \qquad p \not\rightarrow v}{E, \text{let } p = S \text{ in } S' \Downarrow \bot}\ \textsc{LetInNoMatch}$$

$$\frac{t@E = v \qquad p_1 \not\rightarrow v \qquad \ldots \qquad p_n \not\rightarrow v}{E, \text{match } t \text{ with} \mid p_1 \to S_1 \mid \ldots \mid p_n \to S_n \text{ end} \Downarrow \bot}\ \textsc{NoMatch}$$

$$\frac{k < \text{arity}(x) = m \leq n \qquad \forall g.\ \neg(\!|x|\!)(v_1, \ldots, v_m, g)}{\lceil x, (v_1, \ldots, v_k) \rceil\ \ v_{k+1}\ \ \ldots\ \ v_n \Downarrow_{\text{app}} \bot}\ \textsc{AppUnspecCont}$$

FIGURE 2.7 – Règles pour $\bot$.

Theorem 4.

$$\frac{\emptyset \vdash S : \tau \qquad S \Downarrow_S v}{v \in V_\tau}$$

La propriété de progrès (ou progress en anglais), affirme que tout terme bien typé qui n'est pas une valeur peut se réduire. Cette propriété n'est pas vraie pour Skel, puisque Skel est partiel. Ainsi, (branch end : τ) est bien typé mais n'admet pas de réduction. En revanche, si l'on ajoute une valeur spéciale $\bot$, ainsi que les règles appropriées, on peut prouver la propriété de progrès. En particulier, toutes les clôtures qui apparaissent sont des clôtures valides, i.e des clôtures compatibles avec closures_Σ. Les règles appropriées sont celles de la figure 2.7, ainsi que des règles de passage au contexte qu'on ne détaille pas ici. Un travail futur pourra être de prouver en Coq cette propriété.

2.8 Interprétation abstraite

Nous présentons dans cette section une interprétation abstraite des sémantiques squelettiques [8]. Pour simplifier cette présentation, on choisira de ne pas s'intéresser aux enregistrements, leur interprétation étant parfaitement similaire à celle des uplets.

2.8.1 Ensemble de valeurs

Comme pour l'interprétation concrète, il nous faut un ensemble $V_\tau^\#$ pour les valeurs de type τ. On exige aussi que cet ensemble dispose d'éléments privilégiés $\bot_\tau$ et $\top_\tau$. On demande que cet ensemble soit ordonné par un ordre $\sqsubseteq_\tau$, et définisse un opérateur commutatif et associatif $\sqcup_\tau$. Finalement, on demande un opérateur de concrétion $\gamma_\tau : V_\tau^\# \to \mathcal{P}(V_\tau)$.

Definition 5. *On dit que le uplet* $(V^\#, \bot, \top, \sqsubseteq, \sqcup, \gamma)$ *est* ***correct*** *par rapport à* V *si*

- $\sqsubseteq$ *est un ordre partiel sur* $V^\#$, $\sqcup$ *est commutatif et associatif.*
- $\forall ab \in V^\#, a \sqsubseteq a \sqcup b$.
- $\forall a \in V^\#, \bot \sqsubseteq a \sqsubseteq \top$.

— $\forall ab \in V^\#, a \sqsubseteq b \to \gamma(a) \subseteq \gamma(b)$.
— $\forall a \in V^\#, \bot \sqcup a = a$.
— $\gamma(\bot) = \emptyset$ *et* $\gamma(\top) = V$.

On suppose donné, pour chaque type non spécifié τ, un uplet $(V_\tau^\#, \bot_\tau, \top_\tau, \sqsubseteq_\tau, \sqcup_\tau, \gamma_\tau)$, correct vis-à-vis de V_τ.

Pour les types spécifiés, ce uplet est défini de la manière suivante (on se contente de la définition intuitive, mais la définition formelle s'en déduit de manière similaire à la section 2.7.1) :

— Si τ est défini comme $\tau = \mid C_1\ \tau_1 \mid \ \ldots\ \mid C_n\ \tau_n$, alors on pose :
  — $V_\tau^\# = V_{\tau_1}^\# \times \cdots \times V_{\tau_n}^\#$.
  — $\bot_\tau = (\bot_{\tau_1}, \ldots, \bot_{\tau_n})$.
  — $\top_\tau = (\top_{\tau_1}, \ldots, \top_{\tau_n})$.
  — $(a_1^\#, \ldots, a_n^\#) \sqsubseteq_\tau (b_1^\#, \ldots, b_n^\#) \Leftrightarrow (a_1^\# \sqsubseteq_{\tau_1} b_1^\# \wedge \cdots \wedge a_n^\# \sqsubseteq_{\tau_n} b_n^\#)$.
  — $(a_1^\#, \ldots, a_n^\#) \sqcup_\tau (b_1^\#, \ldots, b_n^\#) = (a_1^\# \sqcup_{\tau_1} b_1^\#, \ldots, a_n^\# \sqcup_{\tau_n} b_n^\#)$.
  — $\gamma_\tau(a_1^\#, \ldots, a_n^\#) = \{C_1\ v \mid v \in \gamma_{\tau_1}(a_1^\#)\} \cup \cdots \cup \{C_n\ v \mid v \in \gamma_{\tau_n}(a_n^\#)\}$.
— si $\tau = \tau_1 \times \cdots \times \tau_n$, alors on pose :
  — $V_\tau^\# = \overline{V}_{\tau_1}^\# \times \cdots \times \overline{V}_{\tau_n}^\# \cup \{\bot_\tau\}$ où $\overline{V}_\tau'^\# \triangleq V_\tau'^\# \setminus \{\bot'_\tau\}$.
  — $\top$, $\sqcup$ et $\sqsubseteq$ sont définis de la même façon que plus haut.
  — $\gamma_\tau(a_1^\#, \ldots, a_n^\#) = \{(a_1, \ldots, a_n) \mid a_1 \in \gamma_{\tau_1}(a_1^\#), \ldots, a_n \in \gamma_{\tau_n}(a_n^\#)\}$.
— pour $\tau = \tau_1 \to \tau_2$, il faut d'abord quelques définitions préalables.

On définit d'abord les environnements abstraits, de la même manière que pour les environnements concrets :

$$E^\# ::= \epsilon^\# \mid E^\#, x : v^\#.$$

On dit que l'environnement $E^\#$ et l'environnement de typage Γ sont cohérents, et on note $\mathrm{OkEnv}^\#(\Gamma \mid E^\#)$, si les noms de variables sont les mêmes et si les types correspondent. Formellement, on définit inductivement $\mathrm{OkEnv}^\#(\cdot \mid \cdot)$ par les deux règles suivantes :

$$\frac{}{\mathrm{OkEnv}^\#(\epsilon \mid \epsilon^\#)} \qquad \frac{v^\# \in V_\tau^\# \qquad \mathrm{OkEnv}^\#(\Gamma \mid E^\#)}{\mathrm{OkEnv}^\#(\Gamma, x : \tau \mid E^\#, x : v^\#)}.$$

On dit enfin que l'environnement de typage Γ, l'environnement concret E et l'environnement abstrait $E^\#$ sont cohérents, et on note $\mathrm{OkSt}_\Gamma(E; E^\#)$, si $\mathrm{dom}(\Gamma) = \mathrm{dom}(E) = \mathrm{dom}(E') \wedge \forall x \in \mathrm{dom}(E).\ \gamma_{\Gamma(x)}(E(x)) \in E^\#(x)$.

— On définit $\mathrm{Clos}_1{}^\#$ et $\mathrm{Unspec}_1{}^\#$ de la manière suivante :

$$\mathrm{Clos}_1^\#(\tau, \tau') \triangleq \{(p, S, E^\#, \Gamma) \mid (\Gamma, p, S) \in \mathrm{closures}^{(\tau,\tau')}(\Sigma) \wedge \mathrm{OkEnv}^\#(\Gamma \mid E^\#)\}$$
$$\mathrm{Unspec}_1^\#(\tau) \triangleq \{(x, (v_1^\#, \ldots, v_m^\#)) \mid x \in T_u \wedge \mathrm{tspec}(x) = (\alpha_1 \to \cdots \to \alpha_m \to \tau) \wedge v_1^\# \in V_{\alpha_1}^\# \wedge \cdots \wedge v_m^\# \in V_{\alpha_m}^\#)\}.$$

On définit maintenant les ensembles $\mathrm{Clos}^\#$ et $\mathrm{Unspec}^\#$ de manière à ce qu'ils soient finis. On retient donc un seul environnement abstrait par clôture, et un seul ensemble d'arguments par terme non spécifié et par arité. C'est notamment à cet endroit que l'analyse peut perdre de la précision, mais gagne en facilité d'automatisation.

$$\mathrm{Clos}^\#(\tau,\tau') \triangleq \{X \subseteq \mathrm{Clos}_1^\#(\tau,\tau') \mid ((p,S,E^\#,\Gamma),(p,S,E'^\#,\Gamma)) \in X^2 \Rightarrow E^\# = E'^\#\}$$
$$\mathrm{Unspec}^\#(\tau) \triangleq \{X \subseteq \mathrm{Unspec}_1^\#(\tau) \mid \{((x,\overline{v}),(x,\overline{w})) \in X^2 \Rightarrow \overline{v} = \overline{w}\}.$$

on a alors $V^\#_{\tau_1\to\tau_2} = \mathrm{Clos}^\#(\tau_1,\tau_2) \times \mathrm{Unspec}^\#(\tau_1 \to \tau_2)$ avec

— $\bot_\tau = (\emptyset,\emptyset)$,

— $\top_{(\tau_1\to\tau_2)} = (\top^{(\tau_1,\tau_2)}_{\mathrm{Clos}^\#}, \top^{\tau_1\to\tau_2}_{\mathrm{Unspec}^\#})$ avec

$$\top^{(\alpha,\beta)}_{\mathrm{Clos}^\#} = \{(p,S,\Gamma^\top) \mid (\Gamma,p,S) \in \mathrm{closures}^{(\alpha,\beta)}(\Sigma)\}.$$

$$\top^{\beta}_{\mathrm{Unspec}^\#} = \{(x,(\top_{\alpha_1},\dots,\top_{\alpha_m})) \mid \mathrm{tspec}(x) = (\alpha_1 \to \cdots \to \alpha_m \to \beta)\}.$$

où $\Gamma^\top$ est un environnement abstrait, le moins précis possible au sens des $(\sqsubseteq_\tau)_\tau$, tel que $\mathrm{OkEnv}(\Gamma \mid \Gamma^\top)$, défini inductivement de la manière suivante :

$$\epsilon^\top = \epsilon^\# \text{ et } (\Gamma, x:\tau)^\top = \Gamma^\top, x : \top_\tau.$$

— $(A,B) \sqsubseteq_{\tau_1\to\tau_2} (A',B')$ ssi $A \sqsubseteq^{(\tau_1,\tau_2)}_{\mathrm{Clos}^\#} A' \wedge B \sqsubseteq^{\tau_1\to\tau_2}_{\mathrm{Unspec}^\#} B'$ où $\sqsubseteq^{(\tau_1,\tau_2)}_{\mathrm{Clos}^\#}$ et $\sqsubseteq^{\tau}_{\mathrm{Unspec}^\#}$ sont définis de la manière suivante :

  — $A \sqsubseteq^{(\tau_1,\tau_2)}_{\mathrm{Clos}^\#} A'$ ssi

$$\forall (p,S,E^\#,\Gamma) \in A.\ \exists E'^\#.\ \left((p,S,E'^\#,\Gamma) \in A' \wedge \forall x \in \mathrm{dom}(\Gamma).\ E^\#(x) \sqsubseteq_{\Gamma(x)} E'^\#(x)\right).$$

  — $B \sqsubseteq^{\tau}_{\mathrm{Unspec}^\#} B'$ ssi

$$\forall (x,(v_1^\#,\dots,v_n^\#)) \in B.\ \exists v_1'^\# \dots v_n'^\#.\ \left((x,(v_1'^\#,\dots,v_n'^\#)) \in B' \wedge \forall i \in [\![1;n]\!].\ v_i \sqsubseteq_{\tau_i} E'^\#(x)\right).$$

— $(A,B) \sqcup_{\tau_1\to\tau_2} (A',B') = (A \sqcup^{(\tau_1,\tau_2)}_{\mathrm{Clos}^\#} A', B \sqcup^{\tau_1\to\tau_2}_{\mathrm{Unspec}^\#} B')$ où $\sqcup^{(\tau_1,\tau_2)}_{\mathrm{Clos}^\#}$ et $\sqcup^{\tau}_{\mathrm{Unspec}^\#}$ sont définis de la manière suivante :

  — On définit d'abord l'union de deux environnements $E_1^\#$ et $E_2^\#$ tels que $\mathrm{OkEnv}^\#(\Gamma \mid E_1^\#)$ et $\mathrm{OkEnv}^\#(\Gamma \mid E_2^\#)$ de la manière suivante : $\mathrm{dom}(E_1^\# \sqcup E_2^\#) = \mathrm{dom}(\Gamma)$ et pour tout $x \in \mathrm{dom}(\Gamma)$, on a $(E_1^\# \sqcup E_2^\#)(x) = (E_1^\#(x) \sqcup_{\Gamma(x)} E_2^\#(x))$. On définit alors :

—

$$\{(p, S, E^\#, \Gamma) \in A \mid \forall E'^\#.\ (p, S, E'^\#, \Gamma) \notin A'\} \cup \{(p, S, E^\#, \Gamma) \in A' \mid \forall E'^\#.\ (p, S, E'^\#, \Gamma) \notin A\} \cup \{(p, S, E_1^\# \sqcup E_2^\#, \Gamma) \mid (p, S, E_1^\#, \Gamma) \in A \wedge (p, S, E_2^\#, \Gamma) \in A'\}.$$

—

$$B \sqcup^\tau_{\mathrm{Unspec}^\#} B' = \{(x, (v_1^\#, \ldots, v_n^\#)) \in B \mid \forall v_1'^\# \ldots v_n'^\#.\ (x, (v_1'^\#, \ldots, v_n'^\#) \notin B'\} \cup \{(x, (v_1^\#, \ldots, v_n^\#)) \in B' \mid \forall v_1'^\# \ldots v_n'^\#.\ (x, (v_1'^\#, \ldots, v_n'^\#) \notin B\} \cup \{(x, (v_1^\# \sqcup_{\tau_1} v_1'^\#, \ldots, v_n^\# \sqcup_{\tau_n} v_n'^\#)) \mid (x, (v_1^\#, \ldots, v_n^\#)) \in B \wedge (x, (v_1'^\#, \ldots, v_n'^\#)) \in B'\}.$$

— $\gamma_{\tau_1 \to \tau_2}(A, B) \subseteq V_{\tau_1 \to \tau_2}$ est défini de la manière suivante :
  — $\langle p, E, S\rangle \in \gamma_\tau(A, B)$ ssi
  $\exists \Gamma E^\#.\ \mathrm{OkSt}_\Gamma(E; E^\#) \wedge (p, S, E^\#) \in A.$
  — $\lceil x, (v_1, \ldots, v_n)\rceil \in \gamma_\tau(A, B)$ ssi
  $\exists v_1^\# \ldots v_n^\#, v_1 \in \gamma(v_1^\#) \wedge \cdots \wedge v_n \in \gamma(v_n^\#) \wedge (x, (v_1^\#, \ldots, v_n^\#)) \in B.$

Lemma 6. *Pour tout type τ, le uplet $(V_\tau^\#, \bot_\tau, \top_\tau, \sqsubseteq_\tau, \sqcup_\tau, \gamma_\tau)$ est correct vis-à-vis de V_τ.*

Démonstration. On prouve le résultat par récurrence sur le type τ. Le seul cas de base est le cas des types non spécifiés. Il est donc vérifié par hypothèse. On n'a ainsi à prouver que l'hérédité.

Types variants Soit un type variant τ avec pour constructeurs $C_1, \ldots, C_n$ de type respectifs $\tau_1, \ldots, \tau_n$. On suppose pour tous les types τ_i, un uplet $(V_{\tau_i}^\#, \bot_{\tau_i}, \top_{\tau_i}, \sqsubseteq_{\tau_i}, \sqcup_{\tau_i}, \gamma_{\tau_i})$ correct vis-à-vis de V_{τ_i}.

On définit tous les éléments comme plus haut. On vérifie les conditions listées ci-dessus. Chacune des conditions s'ensuit immédiatement de la véracité des conditions pour les types $\tau_1, \ldots, \tau_n$.

Types produits Soit un type $\tau = \tau_1 \times \cdots \times \tau_n$. On suppose pour tous les types τ_i, un uplet $(V_{\tau_i}^\#, \bot_{\tau_i}, \top_{\tau_i}, \sqsubseteq_{\tau_i}, \sqcup_{\tau_i}, \gamma_{\tau_i})$ correct vis-à-vis de V_{τ_i}.

Encore une fois, chacune des conditions s'ensuit immédiatement de la véracité des conditions pour les types $\tau_1, \ldots, \tau_n$.

Types flèches Soit un type $\tau = \tau_1 \to \tau_2$. On vérifie la correction de $V_\tau^\#$:
— $\sqsubseteq_\tau$ est un ordre partiel sur $V_\tau^\#$: on prouve d'abord par hypothèse de récurrence que $\sqsubseteq^{(\tau_1,\tau_2)}_{\mathrm{Clos}^\#}$ et $\sqsubseteq^{(\tau_1\to\tau_2)}_{\mathrm{Unspec}^\#}$ sont des ordres partiels, puis le résultat s'ensuit immédiatement.
— $\sqcup_\tau$ est commutatif et associatif : même idée que pour $\sqsubseteq_\tau$.

— $\forall ab \in V^\#, a \sqsubseteq a \sqcup b$: même idée.
— $\forall a \in V^\#, \bot \sqsubseteq a \sqsubseteq \top$: même idée.
— $\forall a \in V^\#, \bot \sqcup a = a$: même idée.
— $\gamma(\bot) = \emptyset$: immédiat.
— $\gamma(\top) = V_\tau$: par double inclusion. L'inclusion directe étant triviale, on montre l'inclusion réciproque. Soit $x \in V_\tau$. Si $x = \lceil y, (v_1, \ldots, v_n) \rceil$, alors $(y, (\top, \ldots, \top)) \in \top_{\text{Unspec}^\#}$, donc $x \in \gamma(\top)$. De même, si $x = \langle p, E, S \rangle$, alors $(p, S, E^\top) \in \top_{\text{Clos}^\#}$ où $E^\top$ est défini inductivement de la manière suivante :

$$\epsilon^\top = \epsilon^\# \text{ et } (E, x : v)^\top = E^\top, x : \top.$$

Ainsi, $x \in \gamma(\top)$, ce qui conclut la preuve.
— $\forall ab \in V^\#, a \sqsubseteq b \Rightarrow \gamma(a) \subseteq \gamma(b)$: immédiat.

□

2.8.2 Évaluation

L'extension d'environnements est définie de la manière suivante :

$$E + p \mapsto \bot \triangleq \bot$$
$$E + \_ \mapsto v^\# \triangleq E, \text{ si } v^\# \neq \bot$$
$$E + x \mapsto v^\# \triangleq E, x : v^\#, \text{ si } v^\# \neq \bot$$
$$E + C_i\ p \mapsto (v_1^\#, \ldots, v_n^\#) \triangleq E + p \mapsto v_i^\#$$
$$E + (p_1, \ldots, p_n) \mapsto (v_1, \ldots, v_n) \triangleq (E + p_1 \mapsto v_1) \ldots + p_n \mapsto v_n$$

On suppose donnée pour chaque terme non spécifié `val x : τ` un entier $m = \text{arity}^\#(x)$ tel que $\tau = \alpha_1 \to \cdots \to \alpha_m \to \tau'$ et, si $m \neq 0$, une relation (m+1)-aire $(\!|x|\!)^\# \in \mathcal{P}(V_{\alpha_1}^\# \times \cdots \times V_{\alpha_m}^\# \times V_{\tau'}^\#)$. Dans le cas où $m = 0$, on demande $(\!|x|\!)^\# \in V_{\tau'}^\#$.

Voici maintenant les règles pour évaluer une étape des termes en interprétation abstraite. Elles sont de la forme $[\![t]\!](E^\#, \mathcal{T}^\#) \Downarrow^\# v^\#$ où t est un terme, $E^\#$ est un environnement abstrait, $v^\#$ une valeur abstraite, et $\mathcal{T}^\#$ est un ensemble de triplets (terme, environnement, valeurs), représentant les termes déjà évalués.

$$\frac{E^\#(x) = v^\#}{[\![x]\!](E^\#, \mathcal{T}^\#) \Downarrow^\# v^\#}\ \textsc{Var} \qquad \frac{\texttt{val x:}\tau\texttt{ = t} \qquad T^\#(t, \epsilon^\#, v^\#)}{[\![x]\!](E^\#, \mathcal{T}^\#) \Downarrow^\# v^\#}\ \textsc{TermSpec}$$

$$\frac{\texttt{val x : }\tau}{[\![x]\!](E^\#, \mathcal{T}^\#) \Downarrow^\# (\emptyset, \{(x, ())\})}\ \textsc{TermUnspec} \qquad \frac{T^\#(t, E^\#, v^\#)}{[\![(\mathrm{C}_i\, t)]\!](E^\#, \mathcal{T}^\#) \Downarrow^\# \iota_i(v^\#)}\ \textsc{Const}$$

$$\frac{T^\#(t_1, E^\#, v_1^\#) \qquad \dots \qquad T^\#(t_n, E^\#, v_n^\#)}{[\![(t_1, \dots, t_n)]\!](E^\#, \mathcal{T}^\#) \Downarrow^\# (v_1^\#, \dots, v_n^\#)}\ \textsc{Tuple}$$

$$\frac{(p, S, E^\#) \in \mathrm{Clos}^\#}{[\![(\lambda p : \tau \to S)]\!](E^\#, \mathcal{T}^\#) \Downarrow^\# \left(\{(p, S, E^\#)\}, \emptyset\right)}\ \textsc{Clos}$$

Dans la règle Const, C_i est le i-ème constructeur parmi n, et ι_i est défini par $\iota_i(v^\#) \triangleq (\bot_{\tau_1}, \dots, \bot_{\tau_{i-1}}, v^\#, \bot_{\tau_{i+1}}, \dots, \bot_{\tau_n})$.

On donne également les règles pour les squelettes et pour les applications. Les règles pour évaluer les squelettes sont de la forme $[\![S]\!](E^\#, \mathcal{T}^\#, \mathcal{S}^\#, \mathcal{A}^\#) \Downarrow^\# v^\#$ où S est un terme, $E^\#$ est un environnement abstrait, $v^\#$ une valeur abstraite. Les règles pour évaluer les applications sont de la forme $[\![v_1^\#\ \dots\ v_n^\#]\!](\mathcal{T}^\#, \mathcal{S}^\#, \mathcal{A}^\#) \Downarrow^\#_{\mathrm{app}} v^\#$. où $v_1^\#, \dots, v_n^\#, v^\#$ sont des valeurs abstraites.

Dans les deux expressions ci-dessus, $T^\#$ est un ensemble de triplets (terme, environnement, valeurs), représentant les termes déjà évalués, $S^\#$ est un ensemble de triplets (squelettes, environnement, valeurs), représentant les squelettes déjà évalués, et $A^\#$ est un ensemble de paires (liste de valeurs, valeurs), représentant les applications de fonction.

$$\frac{T^\#(t, E^\#, v^\#)}{[\![\mathrm{ret}\, t]\!](E^\#, \mathcal{T}^\#, \mathcal{S}^\#, \mathcal{A}^\#) \Downarrow^\# v^\#}\ \textsc{Ret}$$

$$\frac{S^\#(S_1, E^\#, v_1^\#) \qquad \dots \qquad S^\#(S_n, E^\#, v_n^\#)}{[\![\mathrm{branch}\ \ S_1\ \ \mathrm{or}\ \ \dots\ \ \mathrm{or}\ \ S_n\ \ \mathrm{end}]\!](E^\#, \mathcal{T}^\#, \mathcal{S}^\#, \mathcal{A}^\#) \Downarrow^\# (v_1^\# \sqcup \dots \sqcup v_n^\#)}\ \textsc{Branch}$$

$$\frac{S^\#(S_1, E^\#, v^\#) \qquad S^\#(S_2, E^\# + p \mapsto v^\#, w^\#)}{[\![\mathrm{let}\, p = S_1\, \mathrm{in}\, S_2]\!](E^\#, \mathcal{T}^\#, \mathcal{S}^\#, \mathcal{A}^\#) \Downarrow^\# w}\ \textsc{LetIn}$$

$$\frac{S^\#(S, E^\# + p \mapsto \top_\tau, w^\#)}{[\![\text{let } p : \tau \text{ in } S]\!](E^\#, \mathcal{T}^\#, \mathcal{S}^\#, \mathcal{A}^\#) \Downarrow^\# w^\#}\ \textsc{Exist} \qquad \frac{}{[\![S]\!](\bot, \mathcal{T}^\#, \mathcal{S}^\#, \mathcal{A}^\#) \Downarrow^\# \bot}\ \textsc{BotEnv}$$

$$\frac{T^\#(t_0, E^\#, f^\#) \qquad T^\#(t_1, E^\#, v_1^\#) \qquad \dots \qquad T^\#(t_n, E^\#, v_n^\#) \qquad A^\#((f^\#, v_1\#, \dots, v_n^\#), w^\#)}{[\![t_0\ t_1 \dots t_n]\!](E^\#, \mathcal{T}^\#, \mathcal{S}^\#, \mathcal{A}^\#) \Downarrow^\# w^\#}\ \textsc{App}$$

$$\frac{\begin{array}{c} T^\#(t, E^\#, v^\#) \\ S^\#(S_1, E^\# + p_1 \mapsto v^\#, w_1^\#) \qquad \dots \qquad S^\#(S_n, E^\# + p_n \mapsto v^\#, w_n^\#) \end{array}}{[\![\text{match } t \text{ with} \mid p_1 \to S_1 \mid \dots \mid p_n \to S_n \text{ end}]\!](E^\#, \mathcal{T}^\#, \mathcal{S}^\#, \mathcal{A}^\#) \Downarrow^\# w_1^\# \sqcup \dots \sqcup w_n^\#}\ \textsc{Match}$$

La règle Match prend chaque branche en ignorant les précédentes. C'est-à-dire qu'on ne tient pas compte dans le calcul de la branche $n+1$ du fait que les branches 1 à n n'ont pas été sélectionnées. Cela peut engendrer une perte de précision dans le cas où les motifs se recouvrent, notamment lorsque le dernier motif est le motif universel _.

Si l'on souhaite plus de précision, on peut définir une sémantique où l'extension d'environnement engendre un résidu, qui représente le sous-ensemble de la valeur abstraite qui n'a pas été choisi par l'environnement.

Pour l'application, comme les valeurs abstraites fonctionnelles sont un uplet fini de valeurs, on définit uniquement comment gérer un élément, et comment gérer l'ensemble vide. Par induction, on a bien géré le cas fini quelconque. On traite tout de même à part le cas d'une application sans arguments (ce qui évite de traiter différemment le cas où il reste des arguments et le cas où il n'en reste pas dans les règles suivantes.

On a donc les règles suivantes :

$$\frac{\begin{array}{c} \mathcal{S}^\#(S, E^\# + p \mapsto v_1^\#, w^\#) \qquad \mathcal{A}^\#((w^\#, v_2^\#, \dots, v_n^\#), x^\#) \\ \mathcal{A}^\#(((A, B), v_1^\#, \dots, v_n^\#), y^\#) \end{array}}{[\![(A \cup \{(p, S, E)\}, B)\ v_1^\#\ \dots\ v_n^\#]\!](\mathcal{T}^\#, \mathcal{S}^\#, \mathcal{A}^\#) \Downarrow_{\text{app}}^\# x^\# \sqcup y^\#}\ \textsc{AppLeft}$$

$$\frac{\begin{array}{c} \text{arity}^\#(x) > m + n \qquad y^\# = (\emptyset, \{(x, (v_1^\#, \dots, v_{m+n}^\#))\}) \\ \mathcal{A}^\#(((A, B), v_1^\#, \dots, v_{m+n}^\#), z^\#) \end{array}}{[\![(A, B \cup \{(x, v_1^\#, \dots, v_m^\#)\})\ v_{m+1}^\#\ \dots\ v_{m+n}^\#]\!](\mathcal{T}^\#, \mathcal{S}^\#, \mathcal{A}^\#) \Downarrow_{\text{app}}^\# y^\# \sqcup z^\#}\ \textsc{AppRightCont}$$

$$\frac{\begin{array}{c} m < k = \text{arity}^\#(x) \leq m + n \\ (\!|x|\!)^\#(v_1^\#, \dots, v_k^\#, w^\#) \qquad \mathcal{A}^\#((w^\#, v_{k+1}, \dots, v_{m+n}), y^\#) \\ \mathcal{A}^\#(((A, B), v_1^\#, \dots, v_{m+n}^\#), z^\#) \end{array}}{[\![(A, B \cup \{(x, v_1^\#, \dots, v_m^\#)\})\ v_{m+1}^\#\ \dots\ v_{m+n}^\#]\!](\mathcal{T}^\#, \mathcal{S}^\#, \mathcal{A}^\#) \Downarrow_{\text{app}}^\# y^\# \sqcup z^\#}\ \textsc{AppRightOver}$$

$$\frac{}{[\![f^\#]\!](\mathcal{T}^\#, \mathcal{S}^\#, \mathcal{A}^\#) \Downarrow_{\text{app}}^\# f^\#}\ \textsc{AppZero} \qquad \frac{}{[\![\bot\ v_1^\#\ \dots\ v_n^\#]\!](\mathcal{T}^\#, \mathcal{S}^\#, \mathcal{A}^\#) \Downarrow_{\text{app}}^\# \bot}\ \textsc{AppBot}$$

On a ensuite :

$$\begin{aligned}
\mathcal{H}_t^\#(\mathcal{T}^\#) &= \left\{(t, E^\#, v) \mid [\![t]\!](E^\#, \mathcal{T}^\#) \Downarrow^\# v^\#\right\} \\
\mathcal{H}_S^\#(\mathcal{T}^\#, \mathcal{S}^\#, \mathcal{A}^\#) &= \left\{(S, E, v) \mid [\![S]\!](E^\#, \mathcal{T}^\#, \mathcal{S}^\#, \mathcal{A}^\#) \Downarrow^\# v^\#\right\} \\
\mathcal{H}_{\text{app}}^\#(\mathcal{T}^\#, \mathcal{S}^\#, \mathcal{A}^\#) &= \left\{(v_1^\#, \ldots, v_n^\#), v^\#) \mid [\![v_1^\# \ \ldots \ v_n^\#]\!](\mathcal{T}^\#, \mathcal{S}^\#, \mathcal{A}^\#) \Downarrow_{\text{app}}^\# v^\#\right\} \\
\mathcal{H}^\#(\mathcal{T}^\#, \mathcal{S}^\#, \mathcal{A}^\#) &= (\mathcal{H}_t^\#(\mathcal{T}^\#), \mathcal{H}_S^\#(\mathcal{T}^\#, \mathcal{S}^\#, \mathcal{A}^\#), \mathcal{H}_{\text{app}}^\#(\mathcal{T}^\#, \mathcal{S}^\#, \mathcal{A}^\#))
\end{aligned}$$

Et enfin, on note :

$$(\mathbb{T}^\#, \mathbb{S}^\#, \mathbb{A}^\#) = \bigcup \left\{(\mathcal{T}^\#, \mathcal{S}^\#, \mathcal{A}^\#) \middle| (\mathcal{T}^\#, \mathcal{S}^\#, \mathcal{A}^\#) \subset \mathcal{H}^\#(\mathcal{T}^\#, \mathcal{S}^\#, \mathcal{A}^\#)\right\}$$

2.8.3 Cohérence

On a alors le résultat suivant :

Theorem 7. *Supposons que pour tout terme non-spécifié $x : \tau$, on ait* $\text{arity}^\#(x) = \text{arity}(x)$*, et qu'on ait également :*

- *si* $\text{arity}(x) = 0$*, alors* $(\!|x|\!) \in \gamma_\tau((\!|x|\!)^\#)$.
- *si* $\text{arity}(x) = m > 0$*, alors pour tout* $v_1, \ldots, v_m$*, pour tout* $v_1^\#, \ldots, v_m^\#$*, pour tout* w*, si* $(\!|x|\!)(v_1, \ldots, v_m, w)$*,* $(\!|x|\!)^\#(v_1^\#, \ldots, v_m^\#, w^\#)$ *et* $v_1 \in \gamma(v_1^\#), \ldots, v_m \in \gamma(v_m^\#)$*, alors* $w \in \gamma(w^\#)$.

Alors

- *pour tout squelette* S *et tous environnements* $\Gamma, E, E^\#$ *tels que* $\text{OkSt}_\Gamma(E; E^\#)$*, si* $\Gamma \vdash S : \tau$*,* $\mathbb{S}(S, E, v)$ *et* $\mathbb{S}^\#(S, E^\#, v^\#)$*, alors on a* $v \in \gamma_\tau(v^\#)$.
- *pour tout terme* t *et tout environnement* $\Gamma, E, E^\#$ *tels que* $\text{OkSt}_\Gamma(E; E^\#)$*, si* $\Gamma \vdash t : \tau$*,* $\mathbb{T}(t, E, v)$ *et* $\mathbb{T}^\#(t, E^\#, v^\#)$*, alors on a* $v \in \gamma_\tau(v^\#)$.
- *pour tous types* $\tau, \tau_1, \ldots, \tau_n$*, pour toutes valeurs* $v, (v_1, \ldots, v_n)$ *et* $v^\#, (v_1^\#, \ldots, v_n^\#)$ *telles que* $v_1 \in \gamma_{\tau_1}(v_1^\#)$*, ...,* $v_n \in \gamma_{\tau_n}(v_n^\#)$*, si* $\mathbb{A}((v_1, \ldots, v_n), v)$ *et* $\mathbb{A}^\#((v_1^\#, \ldots, v_n^\#), v^\#)$*, alors on a* $v \in \gamma_\tau(v^\#)$.

L'interprétation abstraite n'étant pas encore formalisée en Coq, nous n'avons pas prouvé en Coq ce résultat de cohérence. Ce sera un travail futur.

Chapitre 3

Necro Lib

> If you can't get rid of the skeletons in your closet, you'd best teach them to dance.
>
> George Bernard Shaw

Pour manipuler des sémantiques squelettiques écrites en Skel, nous proposons Necro, un écosystème qui donne des outils pour les parser, les typer, les manipuler et les exécuter.

Comme le montre la figure 3.1, Necro est un écosystème modulaire pour manipuler des sémantiques squelettiques. Pour faire fonctionner les différents outils de cet ecosystème, et générer des interpréteurs, des débogueurs, etc., il faut fournir une sémantique écrite en langage Skel (voir section 2).

Tous les outils de l'écosystème se basent sur Necro Lib [1], qui fournit une bibliothèque pour parser, typer et manipuler une sémantique squelettique. Cette bibliothèque définit notamment le type de l'arbre de syntaxe abstraite des sémantiques squelettiques (voir section 3.1), une fonction qui parse et type un fichier vers un tel arbre de syntaxe abstraite (voir section 3.2 et section 3.3) et des fonctions de transformation squelettes vers squelettes pour effectuer diverses opérations, par exemple expanser des monades (voir section 3.4).

3.1 AST

On commence par présenter la structure d'arbre de syntaxe abstraite utilisée pour représenter les sémantiques squelettiques. Il nous faudra pour ça présenter comment sont représentées les sémantiques avec le type `skeletal_semantics` (section 3.1.1), puis comment sont représentés les termes et les squelettes (section 3.1.2), et enfin comment sont représentés les types (section 3.1.3). Ces quatre types sont les éléments principaux du fichier skeltypes.mli, défini dans le dépot de Necro Lib.

3.1.1 Le type `skeletal_semantics`

Le type `skeletal_semantics` défini dans Necro Lib reflète la définition formelle des sémantiques squelettiques présentée en figure 2.1. Au lieu de simplement lister les déclarations, elles

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

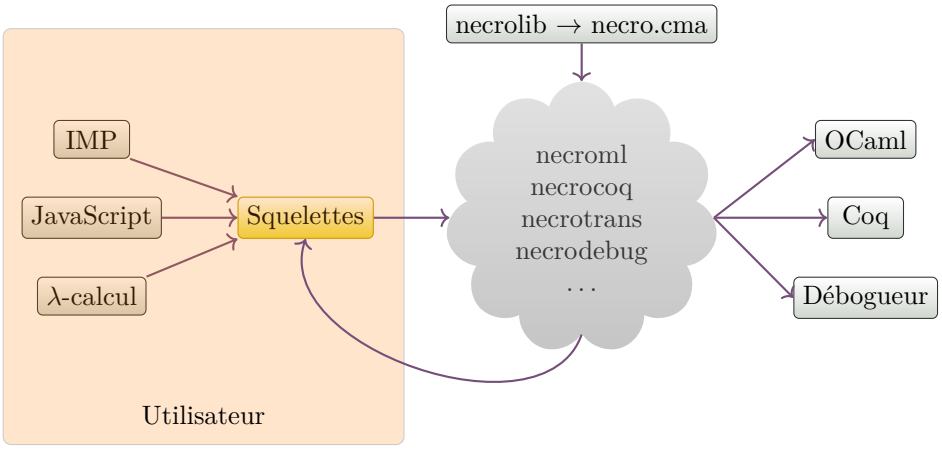

FIGURE 3.1 – L'écosystème Necro.

sont rassemblées dans des tables d'associations, et de la redondance est ajoutée pour manipuler plus simplement les données. Le type des sémantiques squelettiques est défini de la manière suivante :

```
type skeletal_semantics = {
        ss_order : declaration_kind list ;
        ss_types : (comment * necro_type * signature list option) SMap.t ;
        (** Each type name is mapped to its description: the special comment, the
            type signature and the list of constructors (or [None] if the type is
            unspecified. *)
        ss_aliases : (comment * string list * necro_type) SMap.t ;
        (** Each alias name is mapped to its description: the special comment, the
         list of type parameters, and the type referred by the alias *)
        ss_terms : (comment * string list * necro_type * term option) SMap.t ;
        (** Each term name is mapped to its description: the special comment, the
            list of type parameters, the type of the term and optionally, the
            specification of the term. *)
        ss_includes : (skeletal_interface * opened) SMap.t ;
        (** The list of the skeletal semantics of the includes. *)
        ss_binders : (term_kind * from * string * necro_type list) SMap.t
    }
```

Dans cette définition, le type `'a SMap.t`, défini dans le fichier util.ml, est un type de table d'associations dont les clefs sont des chaînes de caractères, et dont les valeurs sont de type `'a`. Reprenons maintenant les champs constitutifs de ce type enregistrement.

— Le champ `ss_order` conserve l'ordre des déclarations dans le fichier. Cet ordre n'a aucune

valeur sémantique, il est uniquement conservé pour pouvoir restituer le fichier dans le même ordre lors du pretty-printing. Cela permet de manipuler une sémantique, et de la réafficher en conservant les choix de structuration qui ont été faits, et donc de garder la même lisibilité. Le type `declaration_kind` permet donc de savoir ce qui est défini. Par exemple, `DTerm "eval"` signifie que c'est la déclaration du terme `eval`. On y conserve aussi certains commentaires, les commentaires spéciaux, qui commencent par `(**`, à la manière de OCamldoc. Ces commentaires pourraient notamment permettre de générer de la documentation, dans le cadre d'un futur outil, ou ils peuvent être utilisés pour donner des commandes spécifiques à des outils. Pour l'instant, aucun outil que nous avons développé ne les utilise.

— Le champ `ss_types` liste les types qui ont été déclarés. S'il s'agit d'un type non spécifié, `None` lui est associé. S'il s'agit d'un type spécifié (à l'exception des alias de types), on donne l'ensemble de ses constructeurs (pour un type variant) ou l'ensemble de ses champs (pour un type enregistrement).

— Le champ `ss_aliases` liste les alias de type et leur définition. Ils sont identifiés séparément des autres types, car ils ne constituent pas un nouveau type à part entière, mais simplement un raccourci d'écriture. On verra d'ailleurs dans la section 3.1.3 qu'ils ne constituent qu'une étiquette dans la définition des types. Le deuxième argument de type `string list` représente la liste des arguments de types pour les alias polymorphes.

— Le champ `ss_terms` liste les termes spécifiés ou non. La troisième composante du quadruplet donne le type déclaré pour le terme, et la dernière composante donne la spécification du terme (ou `None`) si le terme est non-spécifié. Le deuxième argument de type `string list` représente la liste des arguments de types pour les termes polymorphes.

— Le champ `ss_includes` est utilisé pour permettre une approche modulaire à la rédaction de sémantiques, et réutiliser des fichiers communs. Cette approche est présentée dans la section 6.2.

— Enfin, le champ `ss_binders` retient une liste de raccourcis d'écritures pour les lieurs, comme présenté dans la section 2.5.

3.1.2 Les types `term` et `skeleton`

Les types `pattern`, `term` et `skeleton` définis dans Necro Lib reflètent encore une fois la définition formelle des sémantiques squelettiques présentée en figure 2.1.

Voici d'abord la définition du type `pattern` :

```
type pattern =
    | PWild of necro_type (** The pattern [_]. *)
    | PVar of string * necro_type
    | PConstr of constructor * necro_type list * pattern
    (** [PConstr (("C", ty), [ty1; ...; tyn], p)] is the pattern [C p]. [ty] is
```

```
        the output type of [C], and [ty1; ...; tyn] are the inferred type arguments
        for [C]. *)
    | PTuple of pattern list
    | PRecord of (field * necro_type list * pattern) list
    (** [PRecord s] means a record, in which for every [(n, ta, p)] in [s],
        there is a field named [n] whose inferred type arguments are [ta], and
        whose pattern is [p] *)
```

Les motifs `PWild` et `PVar` sont décorés par des types, de sorte que le type de tout motif puisse être déterminé. Les autres constructeurs s'expliquent par eux-mêmes.

Voici maintenant la définition des types `typed_variable`, `term` et `skeleton` :

```
type typed_var =
    | TVLet of term_name * necro_type
    | TVTerm of term_kind * from * term_name * necro_type list * necro_type

type term =
    | TVar of typed_var
    | TConstr of constructor * necro_type list * term
    (** [TConstr (C, [ty1, ..., tyn], t)] is [C<ty1,...,tyn> t].*)
    | TTuple of term list
    | TFunc of pattern * skeleton
    | TField of term * necro_type list * field
    | TRecMake of (field * necro_type list * term) list
    | TRecSet of term * (field * necro_type list * term) list

and skeleton =
    | Branching of necro_type * skeleton list
    (** [Branching (ty, sl)] is a branching with branches [sl] and with the
        branches being of type [ty]. *)
    | Match of term * necro_type * (pattern * skeleton) list
    | LetIn of bind * pattern * skeleton * skeleton
    | Exists of pattern * necro_type * skeleton
    | Return of term
    | Apply of term * term list
```

Une variable typée est ou bien `TVLet`, pour indiquer qu'elle a été créée par une liaison `let in`, ou `TVTerm` pour indiquer qu'elle fait référence à un terme défini au niveau principal du fichier.

On donne ci-dessous un exemple de terme et sa traduction dans l'AST. On supposera que la valeur `nat` est définie en OCaml, de type `necro_type`, et représente le type Skel `nat`. On simplifie légèrement l'AST pour cet exemple, en remplaçant `constructor` par `string`.

```
λ x : nat →
    match x with
    | Succ y →
        y
    | Zero →
        (branch end: nat)
    end
```

```
TFunc (PVar ("x", nat),
    Match (TVar (TVLet ("x", nat)), nat
        [ (PConstr ("Succ", [], PVar ("y", nat)),
            Return (TVar (TVLet ("y", nat))))
        ; (PConstr ("Zero", [], PTuple []),
            Branching (nat, []))
        ])
)
```

Le type pris en deuxième argument par le constructeur `Match` est le type de retour du pattern-matching.

On note les remarques suivantes :

— `TFunc` ne prend pas de type en argument, puisque le motif suffit à calculer le type.

— Le constructeur `TField` sert à la fois à dénoter l'accès à un champ d'un terme de type enregistrement, et l'accès à une composante d'un terme de type produit.

— `Branching` attend un type en argument, ce qui permet de typer tous les squelettes, même les branchements vides. Comme on l'a mentionné en section 2, un branchement vide doit être explicitement typé en Skel.

— `LetIn` prend un argument de type `bind`. En effet, les lieurs personnalisables ne sont pas simplement des méta-notations, mais de véritables éléments de syntaxe.

3.1.3 Le type `necro_type`

Enfin le type `necro_type` dénote les types de Skel. Comme pour les éléments précédents, le type `necro_type` est très proche de la définition formelle donnée en figure 2.1, mais il gère également le polymorphisme. Ainsi, on doit rajouter le cas des variables de type, ce que réalise le constructeur `Variable`. De plus, comme précisé plus haut, on a fait le choix de ne pas considérer les alias de types comme de réels types de bases, mais simplement comme une décoration de type, donc les alias sont dépliés et encapsulés dans le constructeur `Alias`. Enfin, les arguments de type `from` servent à indiquer la provenance d'un type pour le cas des sémantiques squelettiques écrites en plusieurs fichiers (voir section 6.2).

```
type necro_type =
    | Variable of string
    | Base of from * type_name * type_kind * necro_type list
    (** In [Base (_, n, k, ta)], [n] is the name, [ta] the type arguments (for
        polymorphic types), and [k] is the kind of the type ([Unspec], [Record] or
        [Variant]) *)
    | Alias of from * type_name * necro_type list * necro_type
    (** [Alias (f, n, ta, t)], means the type [t], which is referred by its alias
       [f::n<ta>]. *)
```

```
| Arrow of necro_type * necro_type
| Product of necro_type list
```

3.2 Analyse lexical et syntaxique

La première étape de l'analyse du code est l'analyse lexicale puis l'analyse syntaxique. Peu de choses sont à dire sur l'analyseur lexical. Nous avons fait le choix de minimiser le nombre de mots-clefs, il y en a 12 à date de cette thèse. On fait une différence entre les identifiants en minuscule et en majuscule, comme le fait OCaml par exemple. Les identifiants en majuscule sont ensuite utilisés pour les constructeurs uniquement. On a également fait le choix de permettre des commentaires imbriqués.

Ensuite, l'analyseur syntaxique entre en jeu. Il lit le fichier de sémantique et produit un AST non typé, présenté dans le fichier pskeltypes.mli. Il est également assez usuel. Il regroupe toutes les déclarations une par une. Il associe les commentaires spéciaux aux termes, types, constructeurs ou champs auxquels ils sont rattachés. Il découpe ensuite les déclarations suivant leurs types (par exemple, il regroupe les déclarations de types non-spécifiés), tout en conservant l'ordre des déclarations. Il est à noter que l'AST non typé contient des localisations dans le fichier, pour pouvoir donner l'emplacement d'éventuelles erreurs de typage. Le typeur génère une version légèrement augmentée du type `skeletal_semantics`, qui contient également ces localisations. La fonction `parse_and_type` de Necro Lib donne ainsi un AST de type `lskeletal_semantics`. Pour oublier les informations de localisations, et générer un AST plus simple à manipuler, il est possible d'appliquer la fonction `Skeleton.lss_to_ss`, également disponible dans Necro Lib.

Pour simplifier l'analyse syntaxique, et pour qu'elle soit plus simple à anticiper par un humain, on a choisi certaines conventions. Par exemple, les `match` sont toujours finis par un `end`, a contrario d'autres langages comme OCaml par exemple. On a également choisi d'utiliser le mot clef `val` plutôt que `let` pour les déclarations au niveau principal. En effet, OCaml utilise la même notation pour les déclaration de premier niveau et pour les liaisons `let in`. Cette notation crée une confusion pour les personnes connaissant peu le langage, et peut faire que des erreurs soient mal localisées. L'usage du mot-clef `val` permet de découper très simplement les différentes déclarations. Ainsi, elles sont simplements séparées et simplement repérables par l'ensemble des mots-clefs de début de déclaration, à savoir `type`, `val`, `open` et `binder`.

3.3 Typeur

La première étape pour construire l'arbre de syntaxe abstraite décrit dans la section précédente, après l'analyse lexicale et syntaxique, est de typer l'AST produit par l'analyseur syntaxique. Comme mentionné plus haut, le typeur n'inclut pas pour l'instant de mécanisme d'inférence, mais il fait tout de même un peu d'inférence locale dans le cas des lieurs personnalisables. Il doit également procéder à de l'unification avec les motifs (donc dans le cadre des liaisons

`let in` et dans le cas des reconnaissances de motif).

Le fonctionnement du typeur est assez classique et il est séparé en deux parties. On commence par générer un environnement de typage en analysant tout sauf les termes spécifiés, puis une fois l'environnement de typage calculé, on type les termes spécifiés dans cet environnement. La seconde partie se fait dans une monade lecteur qui enregistre l'environnement de typage et les options éventuelles. Ce que la monade enlève en lisibilité — et c'est très mineur, voire même une question de goût, on peut arguer qu'un code monadique est bien plus clair — elle le fait gagner en maintenabilité. En effet, contrairement à une version précédente qui prenait ces informations en argument, la version monadique permet de rajouter des options (par exemple l'inférence de type dans le futur) en se contentant de modifier les parties directement concernées, et le type `store`.

Avant l'ajout de cette monade, on était obligés de propager des arguments supplémentaires pour le typage, et des arguments supplémentaires portant des informations sur les options de typage. À date de cette thèse, il y a une option qui permet de choisir si l'on affiche les avertissements ou non (par exemple en cas de variable non utilisée).

Les fonctions principales sont les suivantes :

— `assign` qui prend un motif et un type et renvoie un environnement qui peut ensuite servir à étendre l'environnement actuel. Elle transforme aussi le motif lui-même d'un type pour les motifs non-typés (`ppattern`) vers un type pour les motifs typés (`pattern`).
`let ve', p_typed = assign tau p in extend_ve ve ve'` est équivalent à $ve + p \mapsto \tau$.
On a choisi de faire l'extension d'environnement en deux étapes pour pouvoir récupérer séparément les variables qui sont créées par le motif `p`, afin de vérifier que toutes les variables définies sont utilisées, et d'avertir l'utilisateurice si ce n'est pas le cas.

— `type_term` qui prend un environnement et un terme, et renvoie un type. Elle transforme aussi le terme du type des termes sortant de l'analyseur syntaxique (`pterm`) vers le type des termes typés (`term`).
`let tau, t' = type_term ve t in ...` est équivalent à $ve \vdash t : \tau$.

— `type_skeleton` qui prend un environnement et un squelette, et renvoie un type. Elle transforme aussi le squelette du type des squelettes sortant de l'analyseur syntaxique (`pskeleton`) vers le type des squelettes typés (`skeleton`).
`let tau, sk' = type_skeleton ve sk in ...` est équivalent à $ve \vdash sk : \tau$.

Dans le cas où on a une reconnaissance de motifs, on fait également des vérifications usuelles pour s'assurer que les motifs ne sont pas redondants, et qu'ils sont bien exhaustifs. On a défini pour cela une fonction `common_refinement` qui, étant donné un ensemble de motifs, donne un découpage du type en motifs disjoints et exhaustifs, de sorte que le découpage soit plus fin que l'ensemble de motifs proposés en entrée.

Plus formellement, si on donne un ensemble de motifs $p_1, \ldots, p_n$, la fonction `common_refinement` renverra un nouvel ensemble de motifs $p'_1, \ldots, p'_m$ tel que pour tout i, j, avec $1 \leq i \leq n, 1 \leq j \leq m$, ou bien les motifs p_i et p'_j sont disjoints, ou bien le motif p'_j est inclus dans le motif p_i. De plus,

les motifs $p'_1, \ldots, p'_n$ sont tous disjoints entre eux, et donnent une partition du type considéré. Il s'agit donc d'un ensemble de motif plus fin que l'ensemble initial, et fournissant une couverture totale du type filtré.

Par exemple, si l'on a défini `type nat = Zero | Succ nat`, alors on aura :

$$\begin{aligned}\text{common\_refinement } &[(\text{Zero}, \_); (\_, \text{Succ}\_)] = \\ &[(\text{Zero}, \text{Succ}\_); (\text{Zero}, \text{Zero}); (\text{Succ}\_, \text{Zero}); (\text{Succ}\_, \text{Succ}\_)].\end{aligned}$$

On obtient donc par raffinement un ensemble P de motifs. Ensuite, il suffit de vérifier que chaque motif de P est pris en compte par le filtrage pour s'assurer de l'exhaustivité. Et pour vérifier la non-redondance, on vérifie que chaque motif du filtrage contient au moins un motif de P qui n'était pas filtré précédemment.

Dans une version précédente de Necro, nous ne proposions pas de reconnaissance de motif à l'aide d'un `match`, il fallait donc passer par un branchement où chaque branche commençait par une liaison destructurante. Nous utilisions alors ces fonctions test d'exhaustivité et de disjonction pour essayer de générer des match en OCaml lorsque cela était possible.

Cet algorithme est naïf mais fonctionnel. Un travail futur sera de le comparer à des algorithmes existants, notamment celui utilisé par OCaml, pour voir comment il pourrait être optimisé.

3.4 Transformations

Necro Lib propose aussi des transformations de squelette à squelette. Elles sont définies dans la librairie necrolib.cma, qui est utilisable pour générer de nouveaux outils (par exemple, Necro Coq utilise une transformation qui supprime les annotations monadiques pour pouvoir générer du code Coq). Ces transformations sont généralement des fonctions mutuellement récursives sur les squelettes et les termes. Plutôt que d'écrire des fonctions récursives, nous fournissons un fichier d'interprétation qui permet d'expliciter le principe de récursion.

3.4.1 Interprétation

Le fichier interpretation.ml définit le type `('t, 's) interpretation` qui explicite le principe de récursion qui génère des élements de type `'t` à partir des termes et des éléments de type `'s` à partir des squelettes. Il définit également les fonctions récursives qui appliquent ce principe de récursion à un terme ou squelette donné. On a donc :

```
val interpret_term : ('t, 's) interpretation -> term -> 't
val interpret_skeleton : ('t, 's) interpretation -> skeleton -> 's
```

Le type `('t, 's) interpretation` est donc un type enregistrement qui contient un champ par constructeur de type et de terme, et dont le type est le miroir de celui du constructeur. Par

exemple, le constructeur `LetIn` est de type `bind * pattern * skeleton * skeleton -> term`. On a donc un champ `letin_interp` du type `bind -> pattern -> 's -> 's -> 't`.

Ce type d'interprétation est utilisé par Necro ML pour générer du code OCaml, par Necro Coq pour générer du code gallina, mais aussi par Necro Lib pour les transformateurs (voir ci-après), et pour définir des fonctions de manipulation simples sur les termes et squelettes (dans le fichier skeleton.ml).

3.4.2 Transformateurs

Des transformations de sémantique squelettique à sémantique squelettique sont définies dans le fichier transformation.ml. La liste concrète avec leurs spécifications est définie dans le fichier d'interface transformation.mli.

La transformation `delete_monads`, par exemple, permet de remplacer tous les lieurs monadiques par des applications explicites de la fonction `bind` associée. La transformation `inline_monads` quant à elle, essaie d'être plus pertinente en remplaçant les lieurs monadiques par le corps de la fonction, et en faisant quelques β-substitutions pour gagner en lisibilité.

Il est à noter que toutes ses transformations ont été écrites dans un souci de préserver la sémantique, mais suivant la sémantique que l'on veut donner aux squelettes, ce résultat ne sera pas systématiquement obtenu. Par exemple, le transformateur `eta_expanse` fais des η-expansions. Il est connu que dans un contexte avec effets de bords, une η-expansion peut changer la sémantique d'un code en retardant le déclenchement de l'effet. Ce n'est pas le cas en interprétation concrète puisque l'interprétation concrète ne permet pas d'effet, mais d'autres interprétations peuvent en permettre.

Nous développons ci-dessous un autre exemple, avec le transformateur `remove_redef`. Ce transformateur renomme certaines variables de manière à éviter les redéfinitions de variables. Ce faisant, il modifie les environnements de typage des clôtures. Or, en sémantique abstraite (voir section 2.8), on confond les clôtures similaires ayant le même environnement de typage. Donc le résultat de la fonction closures_Σ est modifié, et des clôtures auparavant confondues peuvent être séparées, donnant lieu à une analyse plus fine, ou inversement. Par exemple, nous montrons ci-dessous un code avant et après `remove_redef`, et on voit que deux clôtures qui étaient auparavant confondues sont ensuite séparées :

```
val test ((():())) : () =
    let a = λ x : () → x in
    let a = λ x : () → a x in
    let b = λ x : () → a x in
    b ()
```

```
val test ((():())) : () =
    let a = λ x : () → x in
    let a' = λ x : () → a x in
    let b = λ x : () → a' x in
    b ()
```

Dans le premier code, la deuxième et la troisième clôture sont identiques. Dans le second code, nous avons deux environnement de typage différents, car l'environnement du troisième `let in` contient deux variables au lieu d'une, puisque le deuxième `a` a été renommé en `a'`.

De plus, le corps de la clôture (le squelette) n'est plus le même non plus. Cet exemple n'est pas nécessairement critique, et l'interprétation abstraite pourrait facilement être adaptée pour contrer ce souci, en ajoutant des points de programme. Mais c'est un bon exemple pour montrer que les transformateurs ne peuvent pas préserver la sémantique pour toutes les interprétations.

Ces transformations sont donc à utiliser avec prudence et parcimonie si on ne sait pas quelle interprétation sera choisie.

3.4.3 Application

Illustrons les transformateurs avec l'exemple de la monade d'état. On a montré plus haut l'évaluation d'une boucle while avec monade d'état, on la rappelle ici :

```
type st<a> := state → (a, state)

val bind<a, b> (a:st<a>) (f:a → st<b>): st<b> =
    λ s : state →
        let (a, s) = a s in f a s

val ret<a> (a:a): st<a> = λ s : state → (a, s)

val eval_stmt (t:stmt): st<()> =
match t with
| While (cond, t') →
    let f =%bind eval_expr cond in
    let Bool b = f in
    match b with
    | True →
        eval_stmt t' ;%bind
        eval_stmt t
    | False →
        ret<()> ()
    end
| ...
end
```

Après `necrotrans inlinemonads`, `necrotrans eta`, puis `necrotrans inline ret`, on obtient :

```
val eval_stmt (t:stmt): st<()> =
λ state : state →
match t with
```

```
| While (cond, t') →
    let a' = eval_expr cond in
    let (a, s) = a' state in
    let Bool b = a in
    match b with
    | True →
        let a' = eval_stmt t' in
        let (a', s') = a' s in
        eval_stmt t s'
    | False →
        ((), s)
    end
| ...
end
```

Les transformateurs sont assez basiques, mais il est facile d'en ajouter, et ils peuvent être utiles en préalable à d'autres opérations. Par exemple, l'inlining de monades est intéressant pour rendre la sémantique plus lisible avant d'utiliser Necro Debug par exemple (voir section 6.1).

Comme nous l'avons déjà mentionné, les transformateurs sont écrits pour préserver la sémantique vis-à-vis de l'interprétation concrète. En particulier, on pourrait prouver en Coq que la sémantique d'un programme est la même après le transformateur d'η-expansion. Cette propriété est vraie car l'interprétation concrète ne permet pas de parler d'effets de bords. Un travail futur pourra être de formaliser les transformateurs en Coq pour pouvoir prouver leur correction.

Chapitre 4

Necro ML

Trust in Allah, but tie your camel.

Prophète Mahomet

Necro ML [4] est un générateur d'interpréteurs OCaml. Étant donnée une sémantique squelettique, il produit un foncteur OCaml. Ce foncteur prend en argument la spécification OCaml des types et des termes non spécifiés, et il fournit un interpréteur qui peut calculer l'interprétation concrète de n'importe quel squelette.

Skel ne peut pas être plongé superficiellement dans OCaml, puisque OCaml n'a pas d'opérateur pour traduire les branchements (on n'a que la reconnaissance de motif déterministe en OCaml). Donc les types et les termes sont plongés superficiellement, mais les squelettes sont plongés profondément. On utilise une monade d'interprétation qui spécifie entre autres choses comment les squelettes sont représentés, et comment les liaisons let in et les branchements sont calculés.

4.1 Structure du fichier généré

Quand Necro ML est exécuté sur une sémantique squelettique, il génère un fichier OCaml, qui contient plusieurs modules et signatures de modules.

Nous utiliserons dans cette section l'exemple du langage IMP pour montrer comment ce fichier est structuré. Afin de faire apparaître des branchements, on étend ce langage par un opérateur ambigu `Amb: list<expr> → expr`.

En appliquant necroml sur le fichier `while_amb.sk` disponible sur le dépôt de Necro Tests, on obtient un fichier OCaml décrivant un interpréteur. Il est disponible sur le dépôt de Necro ML [1]. Décomposons en plusieurs éléments :

Le fichier généré définit la signature de module `TYPES` qui déclare tous les types non spécifiés. Pour pouvoir générer un interpréteur OCaml, comme nous le montrerons plus tard, il faudra l'instancier avec des types OCaml.

```
(** The unspecified types *)
module type TYPES = sig
```

1. https://gitlab.inria.fr/skeletons/necro-ml/-/blob/master/test/while_amb.ml

```
  type ident
  type lit
  type state
  type vint
end
```

Le fichier définit ensuite le type des monades d'interprétation. On rentrera dans le détail de son fonctionnement en section 4.2. Elle sert à spécifier notamment comment les branches sont évaluées.

```
(** The interpretation monad *)
module type MONAD = sig
  type 'a t
  val ret: 'a -> 'a t
  val bind: 'a t -> ('a -> 'b t) -> 'b t
  val branch: (unit -> 'a t) list -> 'a t
  val fail: string -> 'a t
  val apply: ('a -> 'b t) -> 'a -> 'b t
  val extract: 'a t -> 'a
end
```

Il définit la signature de module `UNSPEC`. Il contient tous les types et les signatures des termes non spécifiés. On est obligés d'inclure les types spécifiés, puisque les signatures des termes non spécifiés peuvent en dépendre. C'est par exemple le cas de `litToVal`.

```
(** All types, and the unspecified terms *)
module type UNSPEC = sig
  module M: MONAD
  include TYPES

  type value =
  | Int of vint
  | Bool of boolean
  and stmt =
  | While of (expr * stmt)
  | Skip
  | Seq of (stmt * stmt)
  | If of (expr * stmt * stmt)
  | Assign of (ident * expr)
  and expr =
  | Var of ident
```

```
  | Plus of (expr * expr)
  | Not of expr
  | Equal of (expr * expr)
  | Const of lit
  | Amb of expr alist
  and boolean =
  | True
  | False
  and 'a alist =
  | Nil
  | Cons of ('a * 'a alist)

  val add: vint * vint -> vint M.t
  val eq: vint * vint -> boolean M.t
  val litToVal: lit -> value M.t
  val read: ident * state -> value M.t
  val write: ident * state * value -> state M.t
end
```

Il définit ensuite le foncteur `Unspec`. Pour éviter d'avoir à écrire les types spécifiés, qui sont obligatoires dans la signature de module `UNSPEC`, on fournit un foncteur qui fait ceci lui-même. Il définit également une valeur par défaut, qui lève l'exception `NotImplemented`, pour toutes les fonctions non spécifiés. Ainsi, si l'on veut seulement tester une partie de l'interpréteur, ou que l'on fait de l'implémentation graduelle, on n'a pas besoin de tout définir immédiatement. On peut ensuite surcharger les fonctions par leurs vraies valeurs, comme on le verra plus loin.

```
(** A default instantiation *)
module Unspec (M: MONAD) (T: TYPES) = struct
  exception NotImplemented of string
  include T
  module M = M

  type value =
  | Int of vint
  | Bool of boolean
  and stmt =
  | While of (expr * stmt)
  | Skip
  | Seq of (stmt * stmt)
  | If of (expr * stmt * stmt)
```

```
  | Assign of (ident * expr)
  and expr =
  | Var of ident
  | Plus of (expr * expr)
  | Not of expr
  | Equal of (expr * expr)
  | Const of lit
  | Amb of expr alist
  and boolean =
  | True
  | False
  and 'a alist =
  | Nil
  | Cons of ('a * 'a alist)

  let add _ = raise (NotImplemented "add")
  let eq _ = raise (NotImplemented "eq")
  let litToVal _ = raise (NotImplemented "litToVal")
  let read _ = raise (NotImplemented "read")
  let write _ = raise (NotImplemented "write")
end
```

Puis le fichier définit la signature de module pour l'interpréteur. Comme on le mentionnera dans la section 6.2, il est possible de définir des sémantiques modulaires. Cette signature de module sera alors nécessaire pour pouvoir parler du fichier, s'il est appelé depuis un autre fichier.

On remarque que les types `'a -> 'b` sont traduits par des types `'a -> 'b M.t`. Ceci s'explique par une traduction dans la catégorie de Kleisli, que nous expliquerons dans la section 4.2.

```
(** The module type for interpreters *)
module type INTERPRETER = sig
  module M: MONAD

  type ident
  type lit
  type state
  type vint

  type value =
  | Int of vint
  | Bool of boolean
```

```
  and stmt =
  | While of (expr * stmt)
  | Skip
  | Seq of (stmt * stmt)
  | If of (expr * stmt * stmt)
  | Assign of (ident * expr)
  and expr =
  | Var of ident
  | Plus of (expr * expr)
  | Not of expr
  | Equal of (expr * expr)
  | Const of lit
  | Amb of expr alist
  and boolean =
  | True
  | False
  and 'a alist =
  | Nil
  | Cons of ('a * 'a alist)

  val add: vint * vint -> vint M.t
  val amb: 'a alist -> 'a M.t
  val eq: vint * vint -> boolean M.t
  val eval_expr: state * expr -> value M.t
  val eval_stmt: state * stmt -> state M.t
  val litToVal: lit -> value M.t
  val neg: boolean -> boolean M.t
  val read: ident * state -> value M.t
  val write: ident * state * value -> state M.t
end
```

Enfin, le fichier définit le foncteur `MakeInterpreter`. Il prend en argument le module qui spécifie les éléments non spécifiés, et il commence par l'inclure dans sa portée.

```
(** Module defining the specified terms *)
module MakeInterpreter (F: UNSPEC) = struct
  include F
```

Ensuite, le fichier généré définit des éléments pour alléger le code généré. Le `let*` évite d'écrire explicitement un `M.bind` à la place de chaque `let-in`, et on génère autant de `applyi` que nécessaire pour remplacer les applications n-aires, qui doivent être décurryfiées, puisque les fonctions

n-aires `a1 → ... → an → b` ont en OCaml le type `a1 -> (... -> (an -> b M.t) M.t) M.t`.

```
let ( let* ) = M.bind

let apply1 = M.apply
```

Ici comme les applications sont déjà curryfiées, l'intérêt est minime. Mais si on avait des applications jusqu'à trois arguments dans le code Skel, on aurait :

```
let apply1 = M.apply
let apply2 f arg1 arg2 =
  let* _tmp = apply1 f arg1 in
  apply1 _tmp arg2
let apply3 f arg1 arg2 arg3 =
  let* _tmp = apply1 f arg1 in
  apply2 _tmp arg2 arg3
```

Ainsi à chaque application n-aire, on a juste à utiliser `applyn`.

L'interpréteur généré commence ensuite par la fonction `amb` qui définit un choix entre les différents éléments de la liste donnée en arguments. On utilise la fonction `M.branch` pour plonger les branchements, et chaque branche est thunkée à l'aide d'un `function () ->` pour éviter son exécution intempestive.

```
let rec amb: 'a. 'a alist -> 'a M.t  =
  function | (Cons (x, l)) ->
  M.branch [
    (function () ->
      M.ret x) ;
    (function () ->
      apply1 amb l)]
  | _ ->
  M.fail ""
```

Nous montrons ensuite la définition de `eval_expr`. Le code donne également la définition de `eval_stmt` et de `neg`. On voit que le code généré est gardé similaire au code initial. Les `let in` destructurants en Skel ne sont pas nécessairement exhaustifs, et si le motif n'est pas reconnu, ils doivent échouer en Skel. Pour cette raison, on les plonge dans OCaml en utilisant un `match` avec un cas pour gérer la réussite du `let in`, et un cas par défaut qui appelle `M.fail`. Cela nous force à utiliser par moment des variables temporaires, et nous éloigne légèrement du code original dans sa traduction OCaml, ce qui est évidemment inévitable.

Par ailleurs, nous avons du expliciter tous les ret qui sont implicites en Skel, mais nécessaires en OCaml pour valider le typage. La perte en lisibilité n'est pas essentielle, puisque le code n'a

pas vocation à être modifié, et il a le même sens que le code Skel correspondant, donc sa lisibilité a uniquement pour but d'augmenter le niveau de confiance.

```
and eval_expr =
  function (s, e) ->
  begin match e with
  | Const i -> apply1 litToVal i
  | Var x -> apply1 read (x, s)
  | Plus (t1, t2) ->
      let* _tmp = apply1 eval_expr (s, t1) in
      begin match _tmp with
      | Int f1 ->
          let* _tmp = apply1 eval_expr (s, t2) in
          begin match _tmp with
          | Int f2 ->
              let* v = apply1 add (f1, f2) in
              M.ret (Int v)
          | _ -> M.fail ""
          end
      | _ -> M.fail ""
      end
  | Equal (t1, t2) ->
      let* _tmp = apply1 eval_expr (s, t1) in
      begin match _tmp with
      | Int f1 ->
          let* _tmp = apply1 eval_expr (s, t2) in
          begin match _tmp with
          | Int f2 ->
              let* v = apply1 eq (f1, f2) in
              M.ret (Bool v)
          | _ -> M.fail ""
          end
      | _ -> M.fail ""
      end
  | Amb l ->
      let* x = apply1 amb l in
      apply1 eval_expr (s, x)
  | Not t ->
      let* _tmp = apply1 eval_expr (s, t) in
      begin match _tmp with
```

```
        | Bool f1 ->
            let* b = apply1 neg f1 in
            M.ret (Bool b)
        | _ -> M.fail ""
        end
    end
```

Le code est généré en deux passes. Une première passe génère du code Caml dans un AST créé pour l'occasion, et la deuxième passe affiche ce code au format textuel. Cela nous permet d'optimiser l'affichage et de minimiser le nombre de parenthèses inutiles. Cela rend également le code plus facilement maintenable.

Le foncteur `MakeInterpreter` permet donc de construire un interpréteur valide, dès lors qu'on lui donne en argument un module de signature `UNSPEC`. Nous montrerons ensuite comment créer un interpréteur pour le langage IMP.

4.2 Monade d'interprétation

Comme nous l'avons expliqué plus haut, on définit une monade d'interprétation pour encapsuler les calculs. Comme expliqué dans [16], les morphismes $A \to^M B$ du type A vers le type B dans la catégorie de Kleisli associés à une monade M sur les types, sont les morphismes $A \to MB$. Ainsi, toutes les signatures du code Skel sont traduites de cette même manière.

Comme on l'a expliqué dans la section 2, les termes sont des valeurs, et les squelettes sont des calculs. Et les calculs doivent pouvoir gérer le non déterminisme dû aux branchements. Les termes de type τ seront donc représentés par des valeurs de types τ, tandis que les squelettes de type τ seront représentés par des valeurs de type τ `M.t`.

On définit la monade d'interprétation de la manière suivante :

```
module type MONAD = sig
    type 'a t
    val ret: 'a -> 'a t
    val bind: 'a t -> ('a -> 'b t) -> 'b t
    val branch: (unit -> 'a t) list -> 'a t
    val fail: string -> 'a t
    val apply: ('a -> 'b t) -> 'a -> 'b t
    val extract: 'a t -> 'a
end
```

L'opérateur `fail` prend en entrée une chaîne de caractères, qui constitue un message d'erreur, et l'opérateur extract est une construction sans signification théorique, qui sert à extraire un résultat de la monade, typiquement pour l'afficher.

L'opérateur apply permet de faire une application. Il est inutile dans la plupart des cas, puisqu'on l'instancie souvent par une application classique, mais il sert si l'on choisit de retarder l'application, comme c'est le cas notamment dans la monade BFS présentée plus loin.

Il y a plusieurs manières d'instancier cette monade. Nous en proposons certaines, mais l'utilisateurice peut également définir sa propre instance.

Les monades proposées et présentées ci-dessous sont disponibles sur le dépôt de Necro ML [2], et comme mentionné plus haut, elles sont accessibles en utilisant `necromonads.cma` si l'on a installé Necro ML.

4.2.1 Monade identité

La monade identité est celle qui est la plus proche d'un plongement superficiel. Pour les branchements, elle essaie chaque branche l'une après l'autre jusqu'à avoir une branche qui donne un résultat valide. Elle est définie de la manière suivante :

```
module ID = struct
    exception Branch_fail of string
    type 'a t = 'a
    let ret x = x
    let rec branch l =
        match l with
        | [] -> raise (Branch_fail "No branch matches")
        | b1 :: bq -> try b1 () with Branch_fail _ -> branch bq
    let fail s = raise (Branch_fail s)
    let bind x f = f x
    let apply f x = f x
    let extract x = x
end
```

Cette monade est la plus simple à utiliser, et elle fonctionne parfaitement dans de nombreux cas. En particulier, elle est la plus pertinente quand on veut générer un interpréteur pour une sémantique parfaitement déterministe.

Cependant, pour des sémantiques non déterministes, on n'obtient qu'un résultat. Et si on étudie par exemple un langage concurrent, on n'aura jamais que l'un des résultats possibles d'exécution.

De plus, si on se rend compte a posteriori que la première branche n'était pas la bonne, on ne peut pas revenir en arrière et prendre une autre branche. Par exemple, dans le code suivant, le squelette `fail ()` lèvera l'exception `Branch_fail`, alors que `()` est une interprétation valide, en utilisant la seconde branche.

2. https://gitlab.inria.fr/skeletons/necro-ml/-/blob/master/necromonads.ml

```
val fail (x:()): () =
    let f =
        branch
            λ _: () → (branch end: ())
        or
            λ _:() → ()
        end
    in f x
```

Cette monade est donc correcte, puisque si elle renvoie un résultat, il est valide. Cependant, elle n'est pas complète, puisqu'elle ne renvoie jamais plus d'un résultat, et elle peut même échouer alors qu'un résultat valide existe.

4.2.2 Monade de liste

La monade de liste est une monade qui permet de collecter chaque résultat possible, au lieu de renvoyer uniquement l'un de ceux-ci. Pour les branchements, elle concatène donc les résultats de chaque branche. La version définie dans le code de Necro ML renvoie une liste avec unicité de l'occurrence de chaque élément. La comparaison se fait avec `Stdlib.compare`, donc la liste peut contenir des expressions non égales, mais avec des représentations différentes. Nous présentons ici une version simplifiée qui ne fait pas cette vérification. Elle est définie de la manière suivante :

```
module List = struct
    type 'a t = 'a list
    let ret x = [x]
    let rec branch l = Stdlib.List.concat_map (fun f -> f ()) l
    let fail _ = []
    let bind l f = Stdlib.List.concat_map f l
    let apply f x = f x
    let extract x =
        begin match x with
        | a :: q -> a
        | [] -> failwith "No result"
        end
end
```

Pour le cas du squelette `fail ()` ci-dessus, elle renverra donc effectivement le résultat attendu, c'est-à-dire `()`. Mais cette monade ne termine que si chaque branche termine, et elle ne renverra de résultat que lorsque toutes les branches auront été évaluées, ce qui peut prendre un temps exponentiellement long. Par exemple, la fonction suivante renverra 2^n fois le même résultat (à savoir `()`) :

```
term f (n: nat) =
    match n with
    | Zero → ()
    | Succ m → branch f m or f m end
    end
```

On pourrait envisager d'ajouter un mécanisme pour enlever les doublons, mais ça ne règlerait que le cas où ce sont les mêmes valeurs qui sont renvoyées.

Cette monade est donc également correcte, mais n'est pas complète non plus. Cependant, si elle termine, elle renvoie bien tous les résultats possibles, ce qui est une sorte de complétude partielle. En revanche, elle peut être peu efficace dans des cas avec beaucoup de branchements.

4.2.3 Monade de continuation

Nous proposons une monade de continuation, qui ne calcule que la première branche, et évite donc l'explosion exponentielle mentionnée ci-dessus, mais qui garde en mémoire les branches non sélectionnées, et offre donc la possibilité de revenir en arrière si l'on rencontre une erreur [9].

Nous présentons la version polymorphe, mais il en existe une version monomorphe qui renvoie unit. Elle fonctionne de la même manière, et est plus simple à lire, mais nécessite de programmer de manière impérative, ce qui est moins propre à OCaml.

Les valeurs plongées dans la monade de continuation sont des calculs auxquels il manque une continuation pour la suite, et une continuation en cas d'erreur. Le type `'b fcont` représente une continuation pour reprendre en cas d'erreur. On n'a donc besoin d'aucune information, à part un éventuel message d'erreur, et on sait produire une valeur de type `'b`. Le type `('a, 'b) cont` représente une continuation avec un trou de type `'a`, qui produit une valeur de type `'b` quand on lui donne une continuation en cas d'erreur. Enfin, une valeur de type `'a` est un calcul, qui pour tout type `'b`, attend une continuation avec un trou de type `'a`, une continuation d'erreur de type `'b`, et est capable de produire une valeur de type `'b`.

```
module ContPoly = struct
    type 'b fcont = string -> 'b
    type ('a,'b) cont = 'a -> 'b fcont -> 'b
    type 'a t = { cont: 'b. (('a,'b) cont -> 'b fcont -> 'b) }
    let ret (x: 'a) = { cont = fun k fcont -> k x fcont }
    let bind (x: 'a t) (f: 'a -> 'b t) : 'b t =
        { cont = fun k fcont -> x.cont (fun v fcont' -> (f v).cont k fcont') fcont }
    let fail s = { cont = fun k fcont -> fcont s }
    let rec branch l = { cont = fun k fcont ->
            begin match l with
            | [] -> fcont "No branch matches"
            | b :: bs -> (b ()).cont k (fun _ -> (branch bs).cont k fcont)
```

```
            end}
    let apply f x = f x
    let extract x = x.cont (fun a _ -> a) (fun s -> failwith s)
end
```

Cette fois encore, l'exécution de `fail ()` donnera le résultat attendu, à savoir `()`. En effet, l'exécution de `fail ()` choisira la première branche, mais gardera la seconde en mémoire dans sa continuation d'erreur. Une fois que `f x` échoue, il ira dépiler la continuation d'erreur, exécutera la seconde branche, et renverra `()`.

De plus, on peut utiliser la continuation pour reprendre l'exécution si on le désire, et calculer les autres valeurs acceptables.

Néanmoins, la monade de continuation n'est toujours pas complète par rapport à l'interprétation concrète, puisqu'elle exécute toujours la première branche d'abord, et risque ainsi d'être bloquée dans une boucle infinie. La fonction ci-dessous en est un exemple.

```
val loop (_:()): () =
    branch
        loop ()
    or
        ()
    end
```

Cette monade est encore une fois correcte et non complète. Mais si elle termine, on peut utiliser la continuation d'erreur pour relancer l'exécution et tenter de récupérer les autres valeurs de retour possible. Si aucune exécution n'est infinie, on obtient alors successivement toutes les valeurs de retour valides. De plus, cette monade est plutôt efficace.

4.2.4 Monade BFS

Pour corriger ce dernier souci, nous proposons une autre monade d'interprétation, baptisée monade BFS (d'après le parcours en largeur d'un graphe, « Breadth-First Search » en anglais). Cette monade effectue une étape dans chaque branche jusqu'à avoir un résultat, puis elle le renvoie. Cette monade est plus lourde, et son utilité est jusqu'à présent surtout théorique. Mais exécutée sur le programme `loop ()` précédent, elle renvoie bien le résultat `()`.

```
module Bfs = struct
    type 'a t =
        | Ret : 'a -> 'a t
        | Bind : ('b t * ('b -> 'a t)) -> 'a t
        | Branch: (unit -> 'a t) Queue.t -> 'a t
        | Apply : ('a -> 'b t) * 'a -> 'b t
    let ret x = Ret x
```

```
    let branch l =
        let q = Queue.create () in
        let () = Stdlib.List.iter (fun x -> Queue.push x q) l in
        Branch q
    let fail _ = branch []
    let bind x f = Bind (x, f)
    let rec eval_step : type a. a t -> a t =
        let open Queue in
        begin function
        (* Evaluation over *)
        | Ret x ->
                failwith "Impossible to evaluate any further"
        | Branch l when is_empty l ->
                failwith "Impossible to evaluate any further"
        (* Evaluation not over *)
        | Bind (Ret x, f) -> f x
        | Bind (Branch l, f) ->
                let branches = Queue.map (fun x () -> Bind (x (), f)) l in
                Branch branches
        | Bind (x, f) -> Bind (eval_step x, f)
        | Apply (f, x) -> f x
        | Branch l ->
                let x = Queue.take l in
                begin match x () with
                | Ret x -> Ret x
                | Branch l' -> Queue.transfer l' l; Branch l
                | w -> Queue.add (fun () -> eval_step w) l; Branch l
                end
        end
    let rec eval: type a. a t -> a =
        begin function
        | Ret x -> x
        | Branch l when Queue.is_empty l -> failwith "No result"
        | y -> let y' = eval_step y in eval y'
        end
    let apply f x = Apply (f, x)
    let extract x = eval x
end
```

Le principe de la monade BFS est qu'elle calcule étape par étape. Lorsqu'il y a une branche,

elle calcule une étape de la première branche, et elle envoie la branche à la fin (ce qui explique l'implémentation des branchements par une file). Lorsque la première branche est elle-même un branchement, on écrase les branchements en poussant toutes les branches du branchement interne à la fin du branchement externe. Par exemple, on a :

```
branch                                          branch
    branch                                          4
        2                                       or
    or                          eval_step           2
        3                       ⟹              or
    end                                             3
or                                              end
    4
end
```

La définition du `bind` est intéressante. Lorsque l'on a un branchement dans le squelette lié par un `let in`, on fait passer le branchement au niveau supérieur. C'est-à-dire que l'on a :

```
                                                branch
let x =                                             let x = 1 in f x
    branch 1 or 2 end           eval_step       or
in f x                          ⟹                  let x = 2 in f x
                                                end
```

Cette monade est correcte et non complète. Si un résultat existe, elle finira toujours par en donner un, mais elle s'arrête après le premier résultat, sans donner de possibilité de reprendre l'exécution. Elle est également très inefficace.

Une idée d'optimisation est de calculer n étapes dans une branche, avec un n qui peut augmenter au cours du temps. Cela limite les manipulations de files, et augmente donc légèrement l'efficacité.

On note également que la monade BFS est la première à ne pas définir la fonction `apply` par une simple application. En effet, puisque l'on souhaite faire un plongement profond des squelettes avec cette monade, on retarde l'application jusqu'à ce qu'on l'applique réellement. Si l'on ne le faisait pas, on pourrait rester bloqué sur une boucle infinie. Par exemple, si l'on avait plongé superficiellement l'application, l'exécution de `test ()` ne terminerait pas dans l'exemple suivant, alors que `()` est un résultat acceptable :

```
val loop (_:()): () =
    loop ()

val test (_:()): () =
    branch
        loop ()
```

```
    or
        ()
    end
```

4.2.5 Monade BFSYield

Enfin, nous proposons une variante de la monade BFS avec une fonction `yield` qui renvoie non seulement un résultat, mais également le calcul restant pour avoir les autres résultats. Elle est correcte et complète, mais elle peut renvoyer plusieurs fois le même résultat s'il est accessible par plusieurs branches différentes. Le code est très proche de la monade précédente :

```
module BfsYield = struct
    type 'a t =
        | Ret : 'a -> 'a t
        | Bind : ('b t * ('b -> 'a t)) -> 'a t
        | Branch: (unit -> 'a t) Queue.t -> 'a t
        | Apply : ('a -> 'b t) * 'a -> 'b t
        | Yield: 'a * 'a t -> 'a t
    let ret x = Ret x
    let branch l =
        let q = Queue.create () in
        let () = Stdlib.List.iter (fun x -> Queue.push x q) l in
        Branch q
    let fail _ = branch []
    let bind x f = Bind (x, f)
    let rec eval_step : type a. a t -> a t =
        let open Queue in
        begin function
        (* Evaluation over *)
        | Ret _ | Yield _ ->
                failwith "Impossible to evaluate any further"
        | Branch l when is_empty l ->
                failwith "Impossible to evaluate any further"
        (* Evaluation not over *)
        | Bind (Ret x, f) -> f x
        | Bind (Branch l, f) ->
                let branches = Queue.map (fun x () -> Bind (x (), f)) l in
                Branch branches
        | Bind (x, f) -> Bind (eval_step x, f)
        | Apply (f, x) -> f x
```

```
        | Branch l ->
                let x = Queue.take l in
                begin match x () with
                | Ret x -> Yield (x, Branch l)
                | Branch l' -> Queue.transfer l' l; Branch l
                | w -> Queue.add (fun () -> eval_step w) l; Branch l
                end
        end
    let rec eval: type a. a t -> a * a t =
        begin function
        | Ret x -> (x, Branch (Queue.create ()))
        | Branch l when Queue.is_empty l -> failwith "No result"
        | Yield (x, t) -> (x, t)
        | y -> let y' = eval_step y in eval y'
        end
    let apply f x = Apply (f, x)
    let extract x = let (v, _) = eval x in v
    let yield x = eval x
end
```

Elle est complète au sens suivant : Si $E, s \Downarrow v$, alors il existe $n, v_1, \ldots, v_{n-1}$ tel que :

```
yield s = (v_1, s_1)
yield s_1 = (v_2, s_2)
...
yield s_{n-1} = (v, s_n)
```

4.2.6 Randomisation de monade

Enfin, pour certaines monades, l'ordre des branches a une incidence. C'est le cas de la monade identité et de la monade de continuation par exemple. Dans ce cas, on peut utiliser le foncteur Rand défini ci-dessous qui modifie l'ordre des branches de manière aléatoire. Ça peut être pertinent lorsque la sémantique écrite en Skel utilise des branchements pour parler d'un comportement aléatoire. On peut alors souhaiter exécuter l'interpréteur plusieurs fois, avec des résultats différents.

```
let () = Random.self_init ()

let shuffle l =
    let lrand = List.map (fun c -> (Random.bits (), c)) l in
    List.sort compare lrand |> List.map snd
```

```
module Rand (M: MONAD) = struct
    include M
    let branch l =
        branch (shuffle l)
end
```

4.2.7 Autres monades

Comme nous l'avons mentionné, il est possible d'instancier la signature de module `MONAD` avec n'importe quelle monade écrite par l'utilisateur. Nous avons notamment expérimenté l'usage de la monade Causality [3] écrite par Simon Castellan. Elle permet de parler de causalité au niveau de la monade d'interprétation. On a pu ainsi écrire un interpréteur pour un langage concurrent très simple, dont le type des programmes est le suivant :

```
type prgm =
    | Write (var, int)
    | Read var
    | Parallel list<prgm>
    | Seq (prgm, prgm)
```

En ajoutant un terme non spécifié pour entrelacer les calculs, on obtient un interpréteur.

On peut envisager également des monades pour des calculs probabilistes ou toute autre application.

4.2.8 Évaluation

Nous présentons ici les temps d'exécution des différentes monades sur différents programmes. On choisit les trois programmes suivants :

— Un programme de calcul de factoriel avec IMP, pour calculer la factorielle de 10, exécuté 10.000 fois,
— Un programme concurrent simple dans un langage concurrent,
— La résolution d'un Sudoku simple.

Les tests sont présentés sur le dépôt gitlab de Necro ML, dans le dossier `evaluation`

Monad	IMP	concurrency	SudoKu
ID	0,92 seconde	fail	fail
List	1,39 seconde	0,82 seconde	0,60 seconde
ContPoly	1,34 seconde	0,0004 seconde	0,40 seconde
BFS	3,18 secondes	7 minutes et 21 secondes	4,18 secondes

3. https://iso.mor.phis.me/publis/Talk_Causality_Nov_2020.pdf

4.3 Instanciation

Pour générer un interpréteur, il faut donc donner une implémentation OCaml pour les types et termes non spécifiés, et choisir une monade. Des exemples fonctionnels sont disponibles en ligne, dans le dossier `test` du dépôt de Necro ML. Nous décrivons ici l'interpréteur du langage IMP avec opérateur ambigu qu'on peut trouver dans ce dossier. Comme pour la section 4.1, on décompose module par module.

On commence par inclure le fichier généré automatiquement par Necro ML.

```
open While_amb
```

On définit le module `SMap` pour des tables d'associations (pour les environnements).

```
module SMap = Map.Make(String)
```

On commence à définir le module `Input`. Il est dépendant de la monade d'interprétation. De cette manière, on peut écrire ce module une seule fois, et l'instancier pour plusieurs monades différentes.

```
module Input(M: MONAD) = struct
```

On définit d'abord un module pour instancier les types non spécifiés. Le système de modules récursifs de OCaml permet de définir des types non spécifiés qui dépendent des types spécifiés, comme c'est le cas ici pour `state` qui dépend de `value`.

```
    module rec T: sig
            type ident = string
            type lit = int
            type vint = int
            type state = Unspec(M)(T).value SMap.t
        end = T
    include T
```

On définit ensuite les termes non spécifiés. Pour cela, on commence par appeler le foncteur `Unspec`. Le résultat du foncteur définit déjà les termes non spécifiés avec des valeurs par défaut. On surcharge donc les termes non spécifiés. Afin de rester modulaire vis-à-vis de `M`, on pense à utiliser `M.ret` pour que les valeurs aient le bon type. De plus, on rattrape les erreurs avec `M.fail`.

```
    include Unspec(M)(T)
    let add (a, b) = M.ret (a + b)
    let eq (a, b) = M.ret (if a = b then True else False)
    let litToVal lit = M.ret (Int lit)
    let read (ident, state): value M.t =
```

```
        begin match SMap.find_opt ident state with
        | None -> M.fail ("unbound value " ^ ident)
        | Some x -> M.ret x
        end
    let write (ident, state, value) =
        M.ret (SMap.add ident value state)
end
```

Enfin, on n'a plus qu'à appeler le foncteur `MakeInterpreter` pour générer un interpréteur du langage IMP. Comme on le disait, et comme on peut le voir, on peut appeler `MakeInterpreter` avec différentes monades.

```
(* Test something deterministic *)
let () =
    let open MakeInterpreter(Input(Monads.ID)) in
    let t =
        Seq (
            Assign ("a", Const 10),
            If (
                Equal(Var "a", Const 9),
                Assign ("x", Const 1),
                Assign ("x", Const 0)
            )
        )
    in
    let s = eval_stmt (initial_state, t) in
    assert(read ("x", s) = Int 0)

let rec has_values l1 l2 = match l1 with
    | [] -> l2 = []
    | a :: q ->
            List.mem a l2 &&
            has_values q (List.filter (fun x -> x <> a) l2)

(* Test something non-deterministic *)
let () =
    let open MakeInterpreter (Input(Monads.List)) in
    let two_or_three = Amb (Cons (Const 2, Cons(Const 3, Nil))) in
    let t =
        Seq (
```

```
        Assign ("a", two_or_three),
        Assign ("b", Plus (Var "a", Var "a"))
    )
in
let s = eval_stmt (initial_state, t) in
let b = List.map (SMap.find_opt "b") s in
assert(has_values [Some (Int 4); Some (Int 6)] b)
```

Un travail futur serait d'ajouter un analyseur syntaxique, potentiellement généré automatiquement, à base de s-expressions, ou au minimum d'ajouter un moyen pour générer et lier des analyseurs syntaxiques à l'interpréteur.

Chapitre 5

Necro Coq

Le coq éloquent crie dès l'œuf.

Proverbe égyptien

Necro Coq est un outil qui permet de plonger automatiquement une sémantique squelettique donnée dans une formalisation Coq. Elle peut ensuite être utilisée pour prouver la correction d'un programme donné, ou des propriétés de langage. Il est disponible en ligne sur le dépôt gitlab associé [19].

Pour l'instant, Necro Coq ne propose que des fichiers d'interprétation concrète. Pour ajouter de l'interprétation abstraite, il suffira cependant de définir un unique fichier Coq, qui sera valable pour n'importe quelle sémantique.

5.1 Structure

Le choix d'un plongement superficiel est sans doute naturel, mais il est souvent moins pertinent. En l'occurrence, plonger superficiellement dans Coq a été tenté lors du projet JSCert. Si la sémantique est fonctionnelle et lisible, la grosse consommation de mémoire des inversions Coq ne permet pas de l'utiliser efficacement. Suite à cette expérience, on a donc plutôt choisi un plongement profond.

Le plongement profond peut également avoir ses limitations, mais il permet une très claire corrélation entre la théorie, notamment ce qui est présenté en Section 2, et la version Coq.

Necro Coq opère donc un plongement profond de Skel. Ce plongement est défini dans le fichier `files/Skeleton.v`, qui est présenté dans la section 5.2.

L'appel de la commande `necrocoq file.sk` permet de générer un fichier qui contient l'arbre de syntaxe abstraite de la sémantique squelettique spécifiée dans `file.sk`.

Un fichier définit ensuite tous les outils pour affirmer le bon typage d'une sémantique squelettique, il s'agit du fichier `files/WellFormed.v`, présenté dans la section 5.3.

La section 5.4 présente la manière dont les valeurs sont représentées en Coq, pour évaluer les squelettes.

Enfin, on fournit différents fichiers qui fournissent des interprétations pour les squelettes et les termes, présentés dans la section 5.5.

La section 5.6 montre des exemples, et la section 5.7 évalue les applications.

5.2 Plongement de Skel

Comme nous l'avons dit plus haut, Necro Coq opère un plongement profond simple. Le fichier `Skeleton.v` définit plusieurs variables, qui sont les données d'une sémantique squelettiques (similaire au uplet Σ défini en section 2.2).

Il définit aussi les constructions des ensembles de bases, encore une fois de manière similaire à la section 2.2, c'est-à-dire la définition des types, des termes, des squelettes, ...

Par exemple, voici la définition des squelettes :

```
Inductive skeleton: Type :=
| skel_branch : type -> list skeleton -> skeleton
| skel_match : term -> type -> list (pattern * skeleton) -> skeleton
| skel_return : term -> skeleton
| skel_apply : term -> list term -> skeleton
| skel_letin : pattern -> skeleton -> skeleton -> skeleton
| skel_exists : pattern -> type -> skeleton -> skeleton.
```

Enfin, il définit le type des sémantiques squelettiques à l'aide d'un champ enregistrement.

```
Record skeletal_semantics :=
    mk_sem {
        (* Base types: specified and non-specified types *)
        s_base_type: list string;
        (* constructors *)
        s_constructor: list string;
        (* record fields *)
        s_field: list string;
        (* Term names *)
        s_unspec_term: list string;
        s_spec_term: list string;
        (* Signature of constructors *)
        s_ctype: dict (list string * type * string) ;
        (* Signature of fields *)
        s_ftype: dict (list string * string * type) ;
        (* Term declarations *)
        s_unspec_term_decl: dict (list string * type) ;
        s_spec_term_decl: dict (list string * type * term) ;
    }.
```

Les champs correspondent globalement aux ensembles définis dans la section 2.2.

La fonction fields n'est pas nécessaire car elle est inférée de la fonction ftype, et de même, on peut distinguer les types variants, construits et non-spécifiés à l'aide des fonctions ftype et

ctype.

Le `list string`, dans le type de `s_ctype` notamment, sert à gérer le polymorphisme. Et les `dict` sont des dictionnaires. Ils sont implémentés avec des listes d'associations. Ainsi, la recherche se fait en temps linéaire. Un travail futur sera d'utiliser une implémentation plus efficace. Le contexte attendu d'usage de Necro Coq est d'utiliser une sémantique fixée (ou plusieurs), où les dictionnaires en question n'ont pas vocation à être modifié. On pourrait donc utiliser des ABR avec une recherche en temps logarithmique par exemple.

5.3 Typage

Le fichier `WellFormed.v` permet de distinguer les squelettes bien formés des autres. Cela permet par exemple d'éliminer les branchements dont toutes les branches n'ont pas le même type, ou les applications invalides. Toutes les variables et tous les branchements sont annotés en Coq par leur type, de manière à simplifier le typage. Cela génère donc de la redondance. Le fichier de bonne formation vérifie ainsi que les types indiqués sont cohérents.

Ce fichier définit des fonctions de substitutions, qui permettent de spécifialiser un terme, un squelette où un type, à des arguments de type donnés.

Il définit ensuite des prédicats inductifs de bonne formation, dont les règles correspondent aux règles énoncées en figure 2.3, et un prédicat `well_formed_semantics` qui définit ce que cela signifie d'être une sémantique squelettique bien formée. Par construction, et sous l'hypothèse de la correction de Necro Lib, une sémantique générée par Necro Coq est nécessairement bien formée.

On a fait le choix de définir la bonne formation par un prédicat, même si la bonne formation est calculable, parce qu'il est plus simple d'utiliser l'hypothèse de bonne formation, – notamment pour prouver la subject reduction, voir plus bas – lorsqu'elle est exprimée par un prédicat, que lorsqu'il s'agit d'une fonction.

Une tâche en cours est de définir une fonction calculable qui renvoie un booléen décidant si la sémantique est bien formée ou non, puis de prouver que cette fonction renvoie vrai si et seulement si le prédicat de bonne formation est valide.

Pour pallier la non-calculabilité du prédicat `well_formed_semantics`, le fichier définit enfin une tactique `well_formed` qui est capable de prouver la bonne formation d'une sémantique quelconque. Cette tactique décompose le but `well_formed_semantics sem` en plusieurs sous-buts, et cela se fait récursivement de manière assez simple grâce à la minimalité de l'AST de Skel. On teste son bon fonctionnement dans le fichier `test/WF/ListsWF.v` sur une sémantique squelettique définissant des listes et des fonctions les manipulant.

5.4 Valeurs

Les valeurs sont définies dans les fichiers d'interprétation. Puisque tous nos fichiers sont des fichier d'interprétation concrète, les valeurs sont identiques entre les fichiers. Le type qui permet de manipuler les valeurs est défini avec 5 constructeurs.

```
Inductive cvalue : Type :=
| cval_base : forall A, A -> cvalue
| cval_constructor : constr -> cvalue -> cvalue
| cval_tuple: list cvalue -> cvalue
| cval_closure: pattern -> skeleton -> list (string * cvalue) -> cvalue.
| cval_unspec: nat -> string -> list type -> list cvalue -> cvalue.
```

Le constructeur `cval_base` permet de représenter les valeurs des types non-spécifié dans n'importe quel type Coq `A`. Par exemple, la valeur 1 dans un type non-spécifié `int` pourra être représentée en mémoire par `cval_base Z 1` (ici, 1 est dans la portée de `Z`, ce qui signifie qu'il est égal à `Zpos xH`). Nous le présentons dans la section 5.4.1.

Les deux constructeurs suivants sont immédiats. Le quatrième est un opérateur pour enregistrer des clôtures, pour évaluer les λ-abstractions. Les trois arguments sont respectivement le motif lié, le corps à évaluer, et l'environnement courant.

Le 5e constructeur est utilisé pour les termes fonctionnels non-spécifiés. Il correspond au constructeur $\lceil x, (v_1, \ldots, v_n) \rceil$ introduit en section 2.7.1. Nous le présenterons dans la section 5.4.2.

5.4.1 Valeurs de base

Pour dénoter les valeurs de base, les premières versions de Necro Coq demandaient de fournir un type Coq pour chaque type non-spécifié.

Cela fonctionne assez bien tant qu'on n'a pas de types non spécifiés qui dépendent de types spécifiés, par exemple des environnement, si les types de valeurs étaient spécifiés. Cela pose également problème quand on parle de types non-spécifiés polymorphes, et qu'on souhaite les instancier avec des arguments de types spécifiés.

On a donc eu l'idée de mettre n'importe quel type, ce qui donne la liberté de mettre même des types dépendant de `cvalue`.

```
Inductive cvalue :=
| cval_base: forall A, A -> cvalue
| ....
```

Cela pose néanmoins un certain nombre de problèmes. L'un d'eux est le suivant : si l'on sait que `cval_base A a = cval_base A b`, on ne peut pas en déduire que `a = b`, sauf si l'on suppose l'injectivité de l'égalité dépendante, ce qui est équivalent à l'axiome K de Streicher et

donc à UIP [1]. Nous n'avons pas rencontré à ce jour de situation où ce problème se soulevait, mais il ne semble pas impossible qu'il arrive.

5.4.2 Valeurs fonctionnelles non-spécifiées

Une dernière question de formalisation doit être traitée. Puisque Skel autorise à déclarer des termes non-spécifiés, on doit pouvoir les interpréter. L'idée la plus naturelle est de demander une `cvalue` pour chaque terme non-spécifié. Seulement, avec les constructeurs actuels, pour une fonction (par exemple l'addition), il faudrait donc donner une clôture, ce qui est exactement équivalent à spécifier la fonction. On perdrait donc le pouvoir de spécification partielle de Skel.

À la place, on demande une relation entre les arguments et les résultats possibles pour chaque fonction non-spécifiée. Par exemple pour l'addition, on donnera la relation $\{([x; y], x + y) \mid x, y \in \mathbb{N}\}$. On note que c'est une relation fonctionnelle puisque l'addition est une relation déterministe.

Les relations ne peuvent pas être représentées par des relations en Coq, pour des raisons de positivité du type des constructeurs. En effet, pour garder des types cohérents, Coq ne permet pas de définir des constructeurs avec des occurences non-strictement positives des types. Ainsi, la définition suivante serait rejetée par Coq :

```
| cval_func: (cvalue -> cvalue) -> cvalue.
```

On propose donc deux solutions pour représenter des relations. La première approche est dénotationnelle. On représente la relation de manière déclarative plutôt que prédicative pour contourner le problème de positivité.

On aurait donc :

```
| cval_func: forall A, (A -> cvalue * cvalue) -> cvalue.
```

Cette approche est complexe et fonctionne mal. En effet, lorsqu'on évalue une fonction, il faut évaluer la totalité de la fonction, ou deviner quels arguments seront utilisés, pour pouvoir déterminer l'ensemble `A` à utiliser.

Une autre solution, qui ne présente pas ces difficultés, est d'utiliser une approche opérationnelle. De la même manière qu'on utilise des clôtures pour les fonctions spécifiées, on décide de retarder l'évaluation pour les termes non-spécifiés. Ainsi, on a le constructeur suivant pour stocker les applications partielles :

```
| cval_unspec: nat -> string -> list type -> list cvalue -> cvalue.
```

Lorsque le premier argument vaut `n`, cela signifie qu'il manque `S n` arguments. On ajoute un, puisqu'il ne peut pas manquer 0 arguments. La liste de types est l'annotation de types pour les termes non-spécifiés polymorphes, et la liste de valeurs et la liste des arguments déjà fournis.

Par exemple, `add` sera interprété par `cval_unspec 1 "add" [] []`, tandis que `add x` sera interprété par `cval_unspec 0 "add" [] [x]`. Enfin, `add x y` n'est pas une application partielle.

1. https://coq.inria.fr/library/Coq.Logic.EqdepFacts.html#lab1172

Donc le squelette sera évalué en allant déplier la relation qui a été fournie pour `add` (ici, on aura donc fourni $\{([x;y], x+y) \mid x, y \in \mathbb{N}\}$).

Un travail en cours est de retarder l'évaluation non seulement des termes non-spécifiés, mais également des termes spécifiés, pour confirmer notre supposition que cela améliorerait la lisibilité.

5.5 Interprétation

Comme nous l'avons mentionné plus haut, seule l'interprétation concrète est à ce jour écrite en Coq. Elle est proposée avec diverses possibilités pour la sémantique de Skel. On présente notamment deux versions à grand pas et une version à petit pas. Ces différentes versions sont prouvées équivalentes en Coq.

5.5.1 Grand pas, version inductive

Le fichier `Concrete.v`[2] fournit une interprétation concrète à grand pas pour les squelettes. Il utilise l'induction de Coq pour définir les relations `interp_skel`, et `interp_term`et `apply` de signatures suivantes :

```
interp_term: env -> term -> cvalue
interp_skel: env -> skeleton -> cvalue
apply: cvalue -> cvalue list -> cvalue
```

Le type `env` est une table d'associations de valeurs. Ces trois relations correspondent aux relations $E@t = v$ (pour `interp_term`), $S, E \Downarrow v$ (pour `interp_skel`), et $v\ v_1\ \ldots\ v_n \Downarrow_{\text{app}} w$ (pour `apply`). Voici la règle pour un `let in` :

```
| i_letin: forall e e' p s1 s2 v w,
    interp_skel e s1 v ->
    add_asn e p v = Some e' ->
    interp_skel e' s2 w ->
    interp_skel e (skel_letin p s1 s2) w
```

La proposition `add_asn e p v = Some e'` affirme que l'environnement e peut être étendu en liant v à p en un nouvel environnement e', c'est-à-dire que $e + p \mapsto v = e'$.

Ce fichier est généralement le fichier le plus pratique à utiliser pour des applications simples. Notamment, on peut prouver la correction d'un programme de factorielle écrit en IMP à l'aide de `Concrete.v`, comme le démontre le dossier `test/certif` du dépôt de Necro Coq.

2. https://gitlab.inria.fr/skeletons/necro-coq/-/blob/master/files/Concrete.v

5.5.2 Grand pas, version itérative

Comme en section 2.7.3, il existe une autre méthode de gérer la récursivité, en choisissant de ne pas utiliser le principe d'induction de Coq. C'est cette méthode qui est implémentée dans le fichier `Concrete_rec.v`[3].

Au lieu de définir directement les relations `interp_term`, `interp_skel` et `apply`, on définit les relations suivantes :

```
next_triple_term:
    forall (T:env -> term -> cvalue -> Prop),
        (env -> term -> cvalue -> Prop).

next_triple_skel:
    forall (T:env -> term -> cvalue -> Prop),
    forall (S:env -> skeleton -> cvalue -> Prop),
    forall (A:cvalue -> list cvalue -> cvalue -> Prop),
        (env -> skeleton -> cvalue -> Prop).

next_triple_apply:
    forall (T:env -> term -> cvalue -> Prop),
    forall (S:env -> skeleton -> cvalue -> Prop),
    forall (A:cvalue -> list cvalue -> cvalue -> Prop),
        (cvalue -> list cvalue -> cvalue -> Prop).
```

Puis on définit $\mathcal{H}^n(\emptyset, \emptyset, \emptyset)$ de la manière suivante :

```
Fixpoint iter_n n :=
  match n with
  | O => (fun e sk v => False, fun e t v => False, fun v vl w => False)
  | S n =>
      let '(Ht, Hs, Happ) := iter_n n in
      ( next_triple_term Ht,
        next_triple_skel Ht Hs Happ,
        next_triple_apply Ht Hs Happ)
  end.
```

Enfin, on a :

```
interp_term_rec e t v <-> exists n, interp_term_n n e t v
```

où `interp_term_n` est la première projection de `iter_n`.

3. https://gitlab.inria.fr/skeletons/necro-coq/-/blob/master/files/Concrete_rec.v

Un intérêt majeur de ce fichier est qu'il permet de faire une récurrence forte sur `interp_skel` de manière très simple, en faisant une récurrence forte sur n. Dans le dossier `test/lambda` [4] sur le dépôt de Necro Coq, on prouve la propriété `eval_lv_ind` qui est la propriété d'induction du λ-calcul, présentée en section 5.6.2. Comme les squelettes sont plongés profondément, un pas du λ-calcul est transformé en plusieurs pas dans l'interprétation concrète. On utilise donc une récurrence forte sur l'arbre de dérivation pour pouvoir prouver cette propriété. On discute de cet exemple dans la section 5.6.

5.5.3 Petit pas

Nous proposons ensuite un fichier `Concrete_ss.v` qui donne une évaluation à petit pas. La définition à petit pas est la seule qui permette de parler de comportements coinductifs donc de non-terminaison, et c'est également la plus simple à utiliser pour prouver la subject reduction, comme nous le dirons plus bas.

Comme souvent en petit pas, il faut ajouter des constructeurs pour pouvoir stocker des étapes intermédiaires de l'évaluation. On définit donc des types `ext_term` et `ext_skel` qui représentent les termes et les squelettes en cours d'évaluation. On n'a ainsi besoin que de deux prédicats pour l'évaluation des termes et celle des squelettes, puisque les applications peuvent maintenant être gérées en utilisant des squelettes étendus. On présente les types `ext_term` et `ext_skel` :

```
Inductive ext_term:Type :=
| ret_term: cvalue -> ext_term
| cont_term: env -> term -> ext_term
| ext_term_constr: string -> list type -> ext_term -> ext_term
| ext_term_tuple: list ext_term -> ext_term
| ext_term_field: ext_term -> string -> ext_term
| ext_term_nth: ext_term -> nat -> ext_term
| ext_term_record: list (string * ext_term) -> ext_term
| ext_term_rec_set: ext_term -> list (string * ext_term) -> ext_term
with ext_skel:Type :=
| ret_skel: cvalue -> ext_skel
| cont_skel: env -> skeleton -> ext_skel
| ext_skel_match: env -> ext_term -> list (pattern * skeleton) -> ext_skel
| ext_skel_return: ext_term -> ext_skel
| ext_skel_letin: env -> pattern -> ext_skel -> skeleton -> ext_skel
| ext_skel_apply: list ext_term -> ext_skel
| ext_apply: ext_skel -> list cvalue -> ext_skel.
```

Par exemple, le constructeur `ret_term` sert à enregistrer une valeur déjà calculée. Le constructeur `cont_term` sert à enregistrer un terme en cours de réduction avec son environnement. Le

4. https://gitlab.inria.fr/skeletons/necro-coq/-/blob/master/test/lambda/EvalInd.v#L55

constructeur `ext_apply` permet de retenir une application en cours de calcul.

Pour prouver l'équivalence avec la version inductive à grand pas, on définit aussi dans le dossier `proofs/eq_ss` un fichier d'interprétation concrète à grand pas étendus, c'est-à-dire une sémantique proche d'une sémantique grand pas, mais définie pour les constructeurs de la sémantique petit pas.

On prouve alors une chaîne d'implication petit pas → grand pas étendus → grand pas → petit pas. La première implication se prouve via un lemme de concaténation comme dans [23]. La deuxième est immédiate, et la troisième se montre par induction de manière classique, en utilisant des lemmes de plongement.

5.5.4 Machine abstraite

Des interprétations sous formes de machines abstraites ont été définies dans [1]. Elles réutilisent le plongement profond fourni par Necro Coq, et elles sont prouvées correctes par rapport à l'interprétation concrète. L'une d'elles est déterministe et fonctionne de manière similaire à la monade de continuation de Necro ML présentée en section 4.2.3.

5.5.5 Subject Reduction

On prouve que l'interprétation concrète vérifie la propriété de subject reduction vis-à-vis de cette bonne formation. Puisque les interprétations sont équivalentes entre elles, il suffit de le prouver pour `Concrete_ss.v` :

```
Theorem subject_reduction_skel:
    forall sk sk' ty,
        type_ext_skel sk ty ->
        interp_skel_ss sk sk' ->
        type_ext_skel sk' ty.
```

Cette formule Coq se traduit par :

$$\frac{S : \tau \qquad S \to S'}{S' : \tau}$$

où S et S' sont des squelettes étendus.

Cela étant prouvé, et puisque `Concrete_ss.v` et `Concrete.v` sont équivalents, on a :

$$\frac{\emptyset \vdash S : \tau \qquad S \Downarrow_S v}{v \in V_\tau}$$

.

5.6 Des exemples pratiques

5.6.1 Preuve d'un calcul de factoriel

Nous présentons dans cette section un exemple concret, la preuve d'un programme de calcul de factorielle dans le langage IMP. La version complète du code est disponible en ligne, sur le

dépôt de Necro Coq, dans le dossier `test/certif`.

On prend le langage IMP déjà défini et utilisé plus haut. Ses expressions et ses instructions sont donc définies de la manière suivante :

```
type expr =
| Const lit
| Var ident
| Plus (expr, expr)
| Equal (expr, expr)
| Not expr
```

```
type stmt =
| Skip
| Assign (ident, expr)
| Seq (stmt, stmt)
| If (expr, stmt, stmt)
| While (expr, stmt)
```

On définit la totalité de la sémantique à travers deux termes, `eval_expr: state → expr → value` et `eval_stmt: state → stmt → state`.

On définit ensuite le code qui calcule une factorielle. Dans le code que nous avons choisi, nous avons inséré des assignations initiales pour les variables. De cette manière, l'invariant de boucle s'exprime simplement comme une variation de l'état mémoire, et celui-ci a la même structure avant d'avoir fait un tour de boucle. Le code est le suivant :

```
fact := 1;
n := m;
a := 0;
i := 0;
while ¬(n = 0) do
  a := fact;
  i := 1;
  while ¬(i = n) do
    fact := fact + a;
    i := i + 1
  done;
  n := n + (−1)
done
```

À la fin de chaque tour de la boucle principale, n est décrémenté de 1, et la variable `fact` contient $m!/(n)!$. La boucle principale multiplie donc `fact` par `n` en l'additionnant à lui-même `n` fois, grâce à la boucle intérieure.

Le fichier est traduit en un fichier Coq `While.v` grâce à Necro Coq. Ensuite, on passe à la preuve en elle-même. Déjà, on a des termes non-spécifiés, que l'on spécifie grâce aux lignes 130 à 148 :

```
Definition unspec_term_decl_interp: dict (nat * unspec) :=
empty
  + { "add"       → (2, add_interp) }
```

```
  + { "eq"        → (2, eq_interp) }
  + { "isBool"    → (1, isBool_interp) }
  + { "isFalse"   → (1, isFalse_interp) }
  + { "isInt"     → (1, isInt_interp) }
  + { "isTrue"    → (1, isTrue_interp) }
  + { "litToVal"  → (1, litToVal_interp) }
  + { "minus_one" → (0, minus_one_interp) }
  + { "neg"       → (1, neg_interp) }
  + { "one"       → (0, one_interp) }
  + { "read"      → (2, read_interp) }
  + { "v_a"       → (0, v_a_interp) }
  + { "v_fact"    → (0, v_fact_interp) }
  + { "v_i"       → (0, v_i_interp) }
  + { "v_n"       → (0, v_n_interp) }
  + { "write"     → (3, write_interp) }
  + { "zero"      → (0, zero_interp) }%nat.
```

On donne à chaque fois l'arité du terme, et la relation qui correspond. Chacune de ces relations, de type `unspec`, est définie plus haut dans le code. Montrons par exemple comment sont définis `add_interp` et `v_a_interp` :

```
Inductive add_interp: unspec :=
| add_intro i j:
  add_interp [] [cval_base Z i; cval_base Z j] (cval_base value (Int (i+j))).
```

Le premier argument de `add_interp` est la liste des arguments de type de `add` (ici il n'y en a pas, puisque l'addition est monomorphe). Le deuxième argument est l'ensemble des arguments fournis à la fonction `add`, et le troisième argument est le résultat.

```
Inductive v_a_interp: unspec :=
| v_a_intro:
  v_a_interp [] [] (cval_base string "a").
```

Encore une fois, le premier argument de `v_a_interp` est la liste des arguments de type (il n'y en a pas, puisque `v_a` est monomorphe). Le deuxième argument est l'ensemble des arguments (encore une fois il n'y en a pas puisque `v_a` n'est pas une fonction), et le troisième argument est le résultat. On rappelle que l'interprétation concrète théorique définie plus haute demande une valeur $(\!|\texttt{v\_a}|\!) \in V_{\texttt{ident}}$, et nous donnons ici une relation $(\!|\texttt{v\_a}|\!) \in \mathcal{P}(V_{\texttt{ident}})$. C'est un choix qui a été fait pour faciliter l'écriture, la lecture et l'utilisation des sémantiques dans Necro Coq, par l'homogénéisation entre les variables et les fonctions. L'alternative est possible, on pourrait différencier les deux en utilisant la version décurryfiée. Cette alternative est une option qui serait

très simple à utiliser et très pertinente si on adopte des applications à un seul argument comme discuté dans la section 2.2.

Ensuite, on définit des lemmes auxiliaires puis on définit et on prouve par récurrence le lemme qui donne l'invariant de la boucle interne, et le lemme qui donne l'invariant de la boucle externe. Enfin, on énonce le théorème principal qui dit que le programme calcule $n!$ dans la variable `fact`.

```
Theorem fact_while (n:nat):
    exists x,
        interp_skel (eval_fact_env n) eval_fact (cval_base fstate x)
    /\ rd_fm_state x "fact" = Some (Int (factorial n)).
```

C'est-à-dire qu'il existe un état, tel que le programme s'évalue vers cet état, et dans cet état, la variable `fact` contient la valeur attendue. La fonction `factorial` calcule la factorielle de son argument de manière computationnelle :

```
Fixpoint factorial (n:nat): Z :=
match n with
| O => 1
| S n => Z.of_nat (S n) * (factorial n)
end.
```

5.6.2 Preuve d'équivalence de sémantiques

Dans le cadre d'un stage, Olivier Idir a écrit la sémantique petit pas et la sémantique grand pas d'une extension du λ-calcul. Puis il a prouvé l'équivalence de ces deux sémantiques en utilisant Necro Coq. C'est notamment dans ce cadre que nous avons défini le fichier `Concrete_rec.v`.

La version complète du code est disponible en ligne, sur le dépôt de Necro Coq, dans le dossier `test/lambda`.

On commence donc par définir le fichier de sémantique squelettique. Il s'agit donc d'un λ-calcul augmenté avec des entiers. Le type des expressions est défini de la manière suivante :

```
type expression =
| Z
| S expression
| Var variable
| App (expression, expression)
| Fun (variable, expression)
| Match_nat (expression, expression, expression)
```

Les constructeurs `Var`, `App` et `Fun` sont les constructeurs standards du λ-calcul. Les constructeurs `Z` et `S` sont le zéro et le successeur. Enfin, le constructeur `Match_nat` sert à détruire les

entiers. Il est défini en grand pas par les deux règles suivantes :

$$\frac{E \vdash e \Downarrow 0 \qquad E \vdash t_Z \Downarrow w}{E \vdash \text{Match\_nat}(e, t_Z, t_S) \Downarrow w}$$

$$\frac{E \vdash e \Downarrow n+1 \qquad E \vdash t_S \Downarrow \text{Clos}(x, t, E') \qquad E' + x \mapsto n \vdash t \Downarrow w}{E \vdash \text{Match\_nat}(e, t_Z, t_S) \Downarrow w}.$$

On définit ensuite deux sémantiques, une à grand pas et une à petit pas, avec des environnements (plutôt qu'avec des substitutions) Pour la sémantique à petit pas, on définit des constructeurs supplémentaires pour les configurations intermédiaires. Notamment on a la configuration `Etat (env, expression)` pour dénoter les expressions en cours d'évaluation, et la configuration `Val value` pour dénoter une expressions dont l'évaluation est terminée.

Les seuls types et termes que l'on a laissés non-spécifiés sont le type `variable` pour les noms de variables et la fonction qui, étant donnée deux noms de variables, décide de leur égalité.

Ensuite on fait la preuve de l'équivalence entre les deux sémantiques. La preuve est découpée dans plusieurs fichiers. Le premier fichier, `Prelim.v`, définit toutes les notations utiles, spécifie les termes et types non-spécifiés, définit quelques lemmes et tactiques auxiliaires. Le deuxième fichier est `EvalInd.v`, il prouve la propriété d'induction du λ-calcul. On se donne donc un prédicat quelconque P qui prend en argument un environnement, un terme, et une valeur pour le résultat. On suppose une hypothèse d'induction pour chaque règle d'inférence de `eval_lv`. Par exemple, pour le constructeur `S`, on a :

```
Variable HS:
    forall e t v,
        P e t (Num v) ->
        P e (S t) (Num (Sv v)).
```

On déduit alors la conclusion du principe d'induction, c'est-à-dire :

```
Lemma eval_lv_ind :
  forall e t v,
    eval_lv e t v ->
    P e t v.
```

Ce principe permet de faire des raisonnements par induction sur l'évaluation grand pas. On pourrait envisager de générer ce principe d'induction. La propriété est relativement simple à formuler, et il nous semble possible que la preuve soit générée automatiquement. C'est un travail futur que de déterminer s'il est possible d'automatiser la preuve, et de faire le travail le cas échéant.

Les deux fichiers suivants sont `CuttingLemmae.v` et `EmbeddingLemmae.v`, qui définissent des lemmes pour relier l'évaluation en sémantique petit et grand pas. Dans `CuttingLemmae.v`, on

donne les lemmes qui permettent de découper une évaluation petit pas en plusieurs étapes. Par exemple, `cut_App1` dit globalement la chose suivante :

Lemma 8 (cut_App1)**.** *Si* $t_1 t_2 \to^k v$*, alors il existe* k_1*,* k_2*,* k_3 *et* v_1*,* v_2 *tels que* $t_1 \to^{k_1} v_1$*,* $t_2 \to^{k_2} v_2$*, et* $v_1 v_2 \to^{k_3} v$*, et* $k = k_1 + k_2 + k_3$*.*

Dans `EmbeddingLemmae.v`, on donne des lemmes qui servent à extraire l'évaluation d'un constructeur. Par exemple, le lemme `in_out_S`, dit que si $c \to^k v$ alors $Sc \to^k Sv$.

Enfin, le dernier fichier, `Certif_lambda.v`, prouve l'équivalence. Le premier théorème, `bs_to_ss`, se prouve en utilisant `eval_lv_ind`. Le deuxième, `ss_to_bs`, se prouve par induction forte. Le dernier, enfin, se contente de mettre en commun les deux premiers théorèmes :

```
Theorem equivalence_ss_lv :
  forall envir expr val,
    eval_lv envir expr val <-> eval_ss_iter (Etat envir expr) (Val val).
Proof.
  split; [apply bs_to_ss|apply ss_to_bs].
Qed.
```

5.7 Applications et facilité d'utilisation

Nous avons considéré plusieurs applications à Necro Coq, dont certaines se trouvent dans le dossier `test` du dépôt gitlab.

Par exemple, nous avons prouvé la correction d'un code IMP qui calcule la fonction factorielle (dossier `test/certif`). Necro Coq a aussi été utilisé pour prouver l'équivalence entre une version à petit pas et une version à grand pas de la sémantique d'un λ-calcul étendu avec des entiers (dossier `test/lambda`). Dans le cadre de cette preuve d'équivalence, Necro Coq a été amélioré par l'ajout de destructeurs et de tactiques pour manipuler plus proprement et plus simplement les squelettes et les termes.

Par exemple, une tactique iLetIn permet de détruire une hypothèse sur l'évaluation d'un squelette `let in` en trois. On montre ci-dessous les buts avant et après l'application de la tactique

iLetIn.

```
isk : interp_skel env (skel_letin p s1 s2) v
========================== (1 / 1)
goal

  => iLetIn isk Haa bound =>

bound: interp_skel env s1 w
Haa : add_asn e p w = Some env'
isk : interp_skel env' s2 v
========================== (1 / 1)
goal
```

De plus, Necro Coq a été utilisé dans [2] pour fournir le cadre formel permettant la génération automatique d'une preuve *a posteriori* de correction de la transformation d'une sémantique grand pas en une sémantique petit pas.

La version actuelle de Necro Coq a encore quelques défauts de lisibilité. Notamment, le plongement profond est un peu verbeux, et des notations pourraient être ajoutées pour aider à la lisibilité des codes générés. C'est également un travail en cours que de définir les meilleures pratiques pour utiliser Necro Coq, et de les mettre en pratique dans les tests et dans les preuves disponibles sur le dépôt.

Chapitre 6

Autres outils et travaux futurs

Si nombreux que soient les travaux finis,
ceux qui restent à faire sont plus
nombreux.

Proverbe africain

6.1 Necro Debug

Necro Debug [18] est un générateur de débogueurs écrit en OCaml. C'est un outil plus récent donc moins abouti que les précédents. Il a été conçu pour que les éléments manquants, c'est-à-dire la spécification des types et des termes non-spécifiés, soient fournis de la même manière que dans Necro ML. Ainsi, il est tout à fait possible de réutiliser les mêmes modules d'instanciations pour Necro ML et pour Necro Debug.

La version initiale proposait une interface sur un terminal. Elle a été abandonnée au profit d'une interface plus adaptée à l'expérience utilisateur, accessible sur un navigateur, avec l'aide de `js_of_ocaml`. On s'est inspiré pour cela de JSExplain [6]. On peut voir un exemple du débogueur sur le site des squelettes [1].

Le fonctionnement en est assez simple. Une sémantique à machine abstraite est définie en OCaml. La sémantique du langage considéré est plongée superficiellement comme dans Necro Coq. Puis la machine abstraite est exécutée sur le squelette qui nous intéresse.

Nous avons de nombreuses perspectives d'améliorations pour Necro Debug. Nous souhaitons ajouter un bouton qui permette de terminer directement l'évaluation d'un terme au lieu de le décomposer. Nous ajouterons également un moyen de lier un analyseur syntaxique pour pouvoir modifier le terme à évaluer, ou en modifier uniquement des parties. Enfin, à la manière de JSExplain [6], nous ajouterons une fenêtre qui présente en simultané le squelette en cours de calcul, dans la sémantique squelettique.

Nous avons récemment modifié la machine abstraite de manière à ce qu'elle retarde l'évaluation des termes spécifiés de la même manière que celle des termes non-spécifiés, en attendant que tous les arguments soit donnés. C'est moralement l'équivalent de la transformation eta de necrotrans, mais appliquée au niveau méta. Cela permet de rendre plus lisible l'exécution de

1. https://skeletons.inria.fr/debugger/index_while.html

codes monadiques, notamment avec une monade d'état ou de continuation, et d'éviter que le code se déplie en entier avant de s'exécuter. En effet, l'exécution d'un code dans une monade d'état commence par créer une clôture qui prend en entrée un état, et renvoie l'état final, c'est-à-dire qu'il spécialise l'interpréteur au programme donné, puis dans un second temps, il prend l'état en compte, et évalue le code.

6.2 Modularité

La version actuelle de Necro Lib permet de découper une sémantique en plusieurs fichiers en utilisant des directives `open file`, et des espaces de noms `file::term`. La génération de OCaml prend également en charge cette modularité. Mais cette fonctionnalité en est à ses premiers balbutiements, et il reste encore beaucoup de travail à faire, notamment pour la rendre stable et fonctionnelle, mais aussi pour la prendre en charge dans Necro Coq et Necro Debug. Il serait souhaitable également de pouvoir inclure un fichier en spécifiant ses termes non-spécifiés, ou de pouvoir l'inclure plusieurs fois sous plusieurs noms, pour pouvoir par exemple avoir un fichier qui gère les tables d'associations, et de pouvoir l'instancier avec des chaînes de caractères comme clefs dans un point du fichier, et avec des entiers dans un autre point.

Pour continuer le travail de modularité et construire un système cohérent et utilisable, nous pourrons nous inspirer des travaux faits dans le cadre du projet PLanCompS [7], du système de module de OCaml [2], ou encore du système de modules de Why3 [10].

Ensuite, il faudra trouver une manière de transposer cette modularité pour chacune des extractions, d'une manière propre à chaque langage, qui corresponde aux usages pour les outils respectifs.

Lorsque ces questions seront résolues, il sera possible de proposer une bibliothèque standard avec des fichiers de base qui définissent des types de base et des fonctions sur ses types, à la manière de la bibliothèque standard de OCaml. Nous pourrons également spécifier et prouver les spécification de la bibliothèque standard à l'aide de Necro Coq.

6.3 Utilisation externe de Necro

La plus grande puissance de Necro est la simplicité de créer un outil. Comme nous l'avons mentionné, le langage Skel est très minimal, et Necro Lib rend accessible un AST du langage et des fonctions pour pouvoir manipuler les sémantiques. Cette simplicité est illustrée d'une part par la diversité des outils qui ont été créés dans le cadre de cette thèse, mais surtout par le fait que d'autres aient pu utiliser Necro pour d'autres travaux.

On peut notamment citer :

— JSkel, un travail en cours pour formaliser en Skel la sémantique de JavaScript [12].

2. https://v2.ocaml.org/manual/moduleexamples.html

— Des travaux en cours (non publiés pour l'instant) pour décrire la sémantique d'un langage multi-niveau permettant de faire communiquer un serveur, des appareils, et un internet des objets, réalisés par Adam Khayam.
— La génération d'un transformateur de sémantiques squelettiques grand pas vers des sémantiques squelettiques petit pas [2], qui génère des preuves a posteriori en utilisant Necro Coq.
— Des travaux réalisés pour générer un interpréteur OCaml certifié en passant par Necro Coq [1].
— Des travaux en cours pour générer un analyseur CFA à partir de sémantique squelettiques, à la manière de l'ancien Necro CFA, réalisés par Vincent Rébiscoul.

6.4 Travaux futurs

De nombreux travaux restent à réaliser. Déjà, les outils actuels sont à mettre à jour et à améliorer, nous avons décrit au sein de cette thèse un grand nombre d'améliorations possibles.

Un manuel est en cours d'écriture, pour utiliser et comprendre le langage Skel [3]. Un travail futur très important est de finir une première version publiable de ce manuel pour les personnes qui approchent Skel et Necro. Nous publierons également un didacticiel, définissant une version du λ-calcul en Skel, et utilisant différents outils, comme Necro ML pour l'exécuter, Necro Coq pour prouver des propriétés, etc. On pourra se baser sur les travaux entrepris par Olivier Idir [4].

Nous souhaitons également continuer le travail qui a été fait pour formaliser des langages en Skel, par exemple pour formaliser du python, en utilisant les travaux de Raphaël Monat [5].

3. https://gitlab.inria.fr/skeletons/necro-man/
4. https://gitlab.inria.fr/skeletons/necro-coq/-/tree/master/test/lambda
5. https://rmonat.fr/publication/thesis/

Chapitre 7

Conclusion

Tout est accompli

Jn, 19 :30

Définir la sémantique d'un langage de programmation est une chose essentielle, comme nous l'avons expliqué en introduction. Mais ce n'est généralement pas une chose aisée. Un langage est généralement complexe, avec de nombreuses fonctionnalités. Il est également en évolution, avec de nouvelles fonctionnalités ajoutées fréquemment, qui peuvent changer profondément la manière dont le langage est pensé.

Skel et Necro visent à faciliter le travail de description de la sémantique d'un langage.

— Comme nous l'avons montré, le langage Skel est conçu pour être simple à comprendre, flexible et puissant. L'utilisation de monades bien choisies permet de rendre une sémantique lisible et modulaire.
— L'outil Necro ML permet d'extraire un interpréteur à partir d'une sémantique, pour tester le bon fonctionnement de la sémantique, mais aussi pour avoir un interpréteur dont on sait qu'il correspond exactement à la définition formelle du langage.
— L'outil Necro Coq permet d'extraire une formalisation Coq en vue de faire des preuves formelles de correction de programme.

Necro est un écosystème encore jeune, qui pourra être augmenté de nouveaux outils à l'avenir, et dont les outils ont encore de nombreuses perspectives d'amélioration. Necro est simple à prendre en main et à augmenter, comme l'a prouvé cette thèse, où nous avons pu, en l'espace de 3 ans, définir notamment un générateur d'interpréteur, un générateur de formalisation Coq et un générateur de débogueur, et les mettre à jour plusieurs fois, pour correspondre aux évolutions du langage Skel.

On a pu vérifier la facilité d'utilisation de Skel et Necro à plusieurs critères :

— Skel a été utilisé pour décrire un langage complexe, JavaScript, dans le cadre du projet JSkel [12].
— Skel et Necro ML ont été utilisés pour un cours de M2 de sémantique avancé à l'université de Rennes 1.
— Skel et Necro ont intéressé des personnes d'autres laboratoires de recherche qui nous ont demandé des informations dans le cadre de travaux les utilisant.

Necro et Skel offrent des perspectives de recherche riches, et leur utilisation, notamment une

fois que la gestion de la modularité présentée en section 6.2 sera plus aboutie, pourra aider à garantir plus de sécurité et de sûreté dans les programmes et les langages de programmation.

L'objectif de la thèse était de développer un langage et des outils pour permettre de formaliser et de manipuler des sémantiques de langages de programmation. On pourra considérer que l'objectif sera pleinement atteint lorsqu'il sera utilisé par plus de personnes et que les gens créeront leurs propres outils en utilisant Necro Lib.

De nombreuses applications sont encore possibles et assez différentes les unes des autres. Un travail est en cours pour proposer un greffon vim pour aider à écrire des sémantiques squelettiques, on peut également envisager un IDE graphique pour les personnes qui le désirent, ou de développer un Language Server Protocol pour pouvoir intégrer Skel à des IDE existants.

Annexe A

Annexes

It ain't over 'til it's over

Lenny Kravitz

A.1 λ-calcul sans stratégie d'évaluation (avec substitution)

```
type ident
type term =
| Var ident
| App (term, term)
| Lam (ident, term)

val ss (t:term): term =
    match t with
    | Var x -> (branch end: term)
    | Lam (x, body) ->
        let b' = ss body in
        Lam (x, b')
    | App (t1, t2) ->
        branch
            let t1' = ss t1 in
            App (t1', t2)
        or
            let t2' = ss t2 in
            App (t1, t2')
        or
            let Lam (x, body) = t1 in
            subst x body t2 (* body[x←t2] *)
        end
    end
```

```
type bool = | True | False
val eq: ident → ident → bool
val fresh: term → ident (* returns an ident which is not in the term *)

val subst (x:ident) (t1:term) (t2:term): term = (* t2[x←t1] *)
    match t2 with
    | Var y → (* y[x ← t1] *)
            let b = eq x y in
            match b with
            | True → (* x[x ← t1] *) t1
            | False → (* y[x ← t1] *) Var y
            end
    | App (u, v) → (* (u v)[x ← t1] *)
            let u' = subst x t1 u in
            let v' = subst x t1 v in
            App (u', v')
    | Lam (y, t) → (* (λ y · t)[x ← t1] *)
            let b = eq x y in
            match b with
            | True → (* (λ x · t)[x ← t1] *) t2
            | False → (* (λ y · t)[x ← t1] *)
                    (* Handle variable capture *)
                    let z = fresh (App (t1, t)) in
                    let t' = rename y z t in (* subst would work but is less effective *)
                    let t'' = subst x t1 t' in
                    Lam (z, t'') (* No capture since z is free in t1 *)
            end
    end

val rename (x:ident) (y:ident) (t:term): term = (* t[x←y] *)
    match t with
    | Var z →
            let b = eq x z in
            match b with
            | True → Var y
            | False → Var z
            end
    | App (u, v) →
            let u' = rename x y u in
```

```
            let v' = rename x y v in
            App (u', v')
    | Lam (z, t) →
            let b = eq x z in
            let t' = rename x y t in
            match b with
            | True → Lam (y, t')
            | False → Lam (z, t')
            end
    end
```

A.2 Polymorphisme

Nous présentons dans cette annexe la BNF et les règles de typage en incluant le polymorphisme. On notera $\mathcal{TV}$ l'ensemble des variables de type, et $\mathcal{X}$ l'ensemble des noms de variables. On prend $x \in \mathcal{X}$, $a \in \mathcal{TV}$.

Typed X	X^{ta}	::=	$X \mid X < \tau, \ldots, \tau >$
Term	t	::=	$x^{ta} \mid C^{ta}\ t \mid (t, \ldots, t) \mid \lambda p : \tau \to S \mid t.f^{ta} \mid t.i \mid$
			$(f = t, \ldots, f = t) \mid t \leftarrow (f = t, \ldots, f = t)$
Skeleton	S	::=	$t_0\ t_1 \ldots t_n \mid$ **let** $p\ =\ S$ **in** $S \mid$ **let** $p\ :\ \tau$ **in** $S \mid$
			match t **with** "$\mid$"$p \to S \ldots$"$\mid$"$p \to S$ **end** $\mid$ **branch** S **or** $\ldots$ **or** S **end** $\mid t$
Pattern	p	::=	$x \mid \_ \mid C\ p \mid (p, \ldots, p) \mid (f = p, \ldots, f = p)$
Type	τ	::=	$a \in \mathcal{TV} \mid b^{ta} \mid \tau \to \tau \mid \tau \to \tau \mid (\tau, \ldots, \tau)$
Term decl	d_t	::=	**val** $x^{ta} : \tau$ $\mid$ **val** $x^{ta} : \tau = t$
Type decl	d_τ	::=	**type** $b \mid$ **type** $b < \_, \ldots, \_ >\mid$ **type** $b^{ta} =$ "$\mid$" $C\ \tau \ldots$"$\mid$" $C\ \tau \mid$
			type $b^{ta} := \tau \mid$ **type** $b^{ta} = (f : \tau, \ldots, f : \tau)$
File	f_τ	::=	$d_\tau \ldots d_\tau d_t \ldots d_t$

Voici maintenant les règles de typage avec le polymorphisme. Elles sont de la forme $\Gamma \vdash t : \tau$.

Γ est un environnement de typage comme plus haut, qui associe des noms de variables à des types. Ces types peuvent maintenant contenir des variables de type libres, en attente d'être substituées.

Il faut également modifier ctype et ftype qui renvoient maintenant un triplet, dont la première composante est la liste des arguments de type du constructeur (resp. du champ), dans l'ordre où ils sont déclarés, et dont les deux autres sont inchangées. De la même façon, fields renvoie une

paire dont la première composante est la liste des arguments de type du champ.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ \textsc{Var} \qquad \frac{\texttt{val}\ x : \tau}{\Gamma \vdash x : \tau}\ \textsc{TermMonoUnspec} \qquad \frac{\texttt{val}\ x : \tau = t}{\Gamma \vdash x : \tau}\ \textsc{TermMonoSpec}$$

$$\frac{\texttt{val}\ x < a_1, \ldots, a_n >: \tau}{\Gamma \vdash x < \tau_1, \ldots, \tau_n > : \tau\{a_{1..n} \leftarrow \tau_{1..n}\}}\ \textsc{TermPolyUnspec}$$

$$\frac{\texttt{val}\ x < a_1, \ldots, a_n >: \tau = t}{\Gamma \vdash x < \tau_1, \ldots, \tau_n > : \tau\{a_{1..n} \leftarrow \tau_{1..n}\}}\ \textsc{TermPolySpec}$$

$$\frac{\Gamma \vdash t : \tau \qquad \text{ctype(C)} = (\emptyset, \tau, \tau')}{\Gamma \vdash \text{C}\, t : \tau'}\ \textsc{ConstMono}$$

$$\frac{\Gamma \vdash t : \tau\{a_{1..n} \leftarrow \nu_{1..n}\} \qquad \text{ctype(C)} = ((a_1, \ldots, a_n), \tau, \tau')}{\Gamma \vdash \text{C} < \nu_1, \ldots, \nu_n > t : \tau'\{a_{1..n} \leftarrow \nu_{1..n}\}}\ \textsc{ConstPoly}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \qquad \ldots \qquad \Gamma \vdash t_n : \tau_n}{\Gamma \vdash (t_1, \ldots, t_n) : (\tau_1, \ldots, \tau_n)}\ \textsc{Tuple} \qquad \frac{\Gamma + p \mapsto \tau \vdash S : \tau'}{\Gamma \vdash (\lambda p : \tau \to S) : \tau \to \tau'}\ \textsc{Clos}$$

$$\frac{\Gamma \vdash t : \nu \qquad \text{ftype}(f) = (\emptyset, \tau, \nu)}{\Gamma \vdash t.f : \tau}\ \textsc{FieldGetMono}$$

$$\frac{\Gamma \vdash t : \nu\{a_{1..n} \leftarrow \tau_{1..n}\} \qquad \text{ftype}(f) = ((a_1, \ldots, a_n), \tau, \nu)}{\Gamma \vdash t.f < \tau_1, \ldots, \tau_n > : \tau\{a_{1..n} \leftarrow \tau_{1..n}\}}\ \textsc{FieldGetPoly}$$

$$\frac{\Gamma \vdash t : (\tau_1, \ldots, \tau_m) \qquad 1 \leq i \leq m}{\Gamma \vdash t.i : \tau_i}\ \textsc{TupleGet}$$

$$\frac{\text{fields}(\tau) = (\emptyset, \{f_1 : \tau_1, \ldots, f_n : \tau_n\}) \qquad \Gamma \vdash t_1 : \tau_1 \qquad \ldots \qquad \Gamma \vdash t_n : \tau_n}{\Gamma \vdash (f_1 = t_1, \ldots, f_n = t_n) : \tau}\ \textsc{RecMono}$$

$$\frac{\begin{array}{c}\text{fields}(\tau) = ((a_1, \ldots, a_m), \{f_1 : \tau_1, \ldots, f_n : \tau_n\}) \\ \Gamma \vdash t_1 : \tau_1\{a_{1..m} \leftarrow \nu_{1..m}\} \qquad \ldots \qquad \Gamma \vdash t_n : \tau_n\{a_{1..m} \leftarrow \nu_{1..m}\}\end{array}}{\Gamma \vdash (f_1 = t_1, \ldots, f_n = t_n) : \tau\{a_{1..m} \leftarrow \nu_{1..m}\}}\ \textsc{RecPoly}$$

$$\frac{\Gamma \vdash t : \tau \qquad \forall i \in [\![1; m]\!], \Gamma \vdash t_i : \tau_i \qquad \forall i \in [\![1; m]\!], \text{ftype}\, f_i = (\tau_i, \tau)}{\Gamma \vdash t \leftarrow (f_1 = t_1, \ldots, f_m = t_m) : \tau}\ \textsc{FieldSetMono}$$

$$\frac{\Gamma \vdash t : \tau\{a_{1..p} \leftarrow \nu_{1..p}\} \qquad \forall i \in [\![1; m]\!], \Gamma \vdash t_i : \tau_i\{a_{1..p} \leftarrow \nu_{1..p}\} \qquad \forall i \in [\![1; m]\!], \text{ftype}\, f_i = ((a_1, \ldots, a_p), \tau_i, \tau)}{\Gamma \vdash t \leftarrow (f_1 = t_1, \ldots, f_m = t_m) : \tau\{a_{1..p} \leftarrow \nu_{1..p}\}}\ \textsc{FieldSetPoly}$$

$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash \text{ret}\, t : \tau}\ \textsc{Ret} \qquad \frac{\Gamma \vdash S_1 : \tau \qquad \ldots \qquad \Gamma \vdash S_n : \tau}{\Gamma \vdash (S_1 \ldots S_n) : \tau}\ \textsc{Branch}$$

$$\frac{\Gamma \vdash S : \tau \qquad \Gamma + p \mapsto \tau \vdash S' : \tau'}{\Gamma \vdash \text{let}\, p = S \,\text{in}\, S' : \tau'}\ \textsc{LetIn} \qquad \frac{\Gamma + p \mapsto \tau \vdash S : \tau'}{\Gamma \vdash \text{let}\, p : \tau \,\text{in}\, S : \tau'}\ \textsc{Exist}$$

$$\frac{\Gamma \vdash t : \tau \qquad \Gamma + p_1 \mapsto \tau \vdash S_1 : \nu \qquad \ldots \qquad \Gamma + p_n \mapsto \tau \vdash S_n : \nu}{\Gamma \vdash \text{match}\, t \,\text{with}\, |p_1 \to S_1| \ldots |p_n \to S_n \,\text{end} : \nu}\ \textsc{Match}$$

$$\frac{\Gamma \vdash t_0 : \tau_1 \to \cdots \to \tau_n \to \tau \qquad \Gamma \vdash t_1 : \tau_1 \qquad \ldots \qquad \Gamma \vdash t_n : \tau_n}{\Gamma \vdash (t_0\ t_1 \ldots t_n) : \tau}\ \textsc{App}$$

BIBLIOGRAPHIE

[1] Guillaume AMBAL, Sergueï LENGLET et Alan SCHMITT. « Certified Abstract Machines for Skeletal Semantics ». In : *Certified Programs and Proofs*. Philadelphia, United States, jan. 2022.

[2] Guillaume AMBAL, Alan SCHMITT et Sergueï LENGLET. *Automatic Transformation of a Big-Step Skeletal Semantics into Small-Step*. Research Report RR-9363. Inria Rennes - Bretagne Atlantique, sept. 2020. URL : https://hal.inria.fr/hal-02946930.

[3] Martin BODIN, Arthur CHARGUERAUD, Daniele FILARETTI, Philippa GARDNER, Sergio MAFFEIS, Daiva NAUDZIUNIENE, Alan SCHMITT et Gareth SMITH. « A Trusted Mechanised JavaScript Specification ». In : *Conference Record of the Annual ACM Symposium on Principles of Programming Languages* 49 (jan. 2014), p. 87-100. DOI : 10.1145/2578855.2535876.

[4] Martin BODIN, Nathanaëlle COURANT, Enzo CRANCE et Louis NOIZET. *Necro Ocaml Generator*. URL : https://gitlab.inria.fr/skeletons/necro-ml.

[5] Martin BODIN, Philippa GARDNER, Thomas JENSEN et Alan SCHMITT. « Skeletal Semantics and their Interpretations ». In : *Proceedings of the ACM on Programming Languages* 44 (2019), p. 1-31. DOI : 10.1145/3290357. URL : https://hal.inria.fr/hal-01881863.

[6] Arthur CHARGUÉRAUD, Alan SCHMITT et Thomas WOOD. « JSExplain : A Double Debugger for JavaScript ». In : *The Web Conference 2018*. Lyon, France, avr. 2018, p. 1-9. DOI : 10.1145/3184558.3185969.

[7] Martin CHURCHILL, Peter D. MOSSES, Neil SCULTHORPE et Paolo TORRINI. « Reusable Components of Semantic Specifications ». In : (2015), p. 132-179. DOI : 10.1007/978-3-662-46734-3_4. URL : https://doi.org/10.1007/978-3-662-46734-3_4.

[8] P. COUSOT et R. COUSOT. « Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints ». In : *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Los Angeles, California : ACM Press, New York, NY, 1977, p. 238-252.

[9] Olivier DANVY et Andrzej FILINSKI. « Abstracting Control ». In : *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*. LFP '90. Nice, France : Association for Computing Machinery, 1990, p. 151-160. ISBN : 089791368X. DOI : 10.1145/91556.91622. URL : https://doi.org/10.1145/91556.91622.

[10] Jean-Christophe Filliâtre et Andrei Paskevich. « Abstraction and Genericity in Why3 ». In : *ISoLA 2021 - 9th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. T. 12476. Rhodes, Greece, oct. 2021. doi : 10.1007/978-3-030-61362-4_7. url : https://hal.inria.fr/hal-02696246.

[11] Gilles Kahn. « Natural Semantics ». In : *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings*. Sous la dir. de Franz-Josef Brandenburg, Guy Vidal-Naquet et Martin Wirsing. T. 247. Lecture Notes in Computer Science. Springer, 1987, p. 22-39. isbn : 3-540-17219-X. doi : 10.1007/BFb0039592. url : https://doi.org/10.1007/BFb0039592.

[12] Adam Khayam. *JSkel, Work in Progress*. url : https://gitlab.inria.fr/skeletons/jskel.

[13] Adam Khayam, Louis Noizet et Alan Schmitt. « JSkel : Towards a Formalization of JavaScript's Semantics ». In : *JFLA 2021 - Journées Francophones des Langages Applicatifs*. Avr. 2021.

[14] Sheng Liang, Paul Hudak et Mark Jones. « Monad Transformers and Modular Interpreters ». In : *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '95. San Francisco, California, USA : Association for Computing Machinery, 1995, p. 333-343. isbn : 0897916921. doi : 10.1145/199448.199528. url : https://doi.org/10.1145/199448.199528.

[15] John McCarthy. « A Basis for a Mathematical Theory of Computation ». In : *Computer Programming and Formal Systems*. Sous la dir. de P. Braffort et D. Hirschberg. T. 26. Studies in Logic and the Foundations of Mathematics. Elsevier, 1959, p. 33-70. doi : https://doi.org/10.1016/S0049-237X(09)70099-0. url : https://www.sciencedirect.com/science/article/pii/S0049237X09700990.

[16] Eugenio Moggi. « Computational lambda-calculus and monads ». In : *Proceedings. Fourth Annual Symposium on Logic in Computer Science* (1988).

[17] Dominic Mulligan, Scott Owens, Kathryn Gray, Tom Ridge et Peter Sewell. « Lem : Reusable Engineering of Real-world Semantics ». In : *ACM SIGPLAN Notices* 49 (août 2014). doi : 10.1145/2628136.2628143.

[18] Louis Noizet. *Necro Debugger*. url : https://gitlab.inria.fr/skeletons/necro-debug.

[19] Louis Noizet. *Necro Gallina Generator*. url : https://gitlab.inria.fr/skeletons/necro-coq.

[20] Louis Noizet et Alan Schmitt. « Formalisation de Sémantiques Squelettiques ». In : *JLFA 2020 - Journées Francophones des Langages Applicatifs*. Gruissan, France, jan. 2020, p. 1-14. url : https://hal.inria.fr/hal-02512485.

[21] *OTT*. url : https://github.com/ott-lang/ott.

[22] Scott Owens, Peter Böhm, Francesco Zappa Nardelli et Peter Sewell. « Lem : A Lightweight Tool for Heavyweight Semantics ». In : *Interactive Theorem Proving*. Sous la dir. de Marko van Eekelen, Herman Geuvers, Julien Schmaltz et Freek Wiedijk. Berlin, Heidelberg : Springer Berlin Heidelberg, 2011, p. 363-369. isbn : 978-3-642-22863-6.

[23] Casper Bach Poulsen et Peter D. Mosses. « Flag-based big-step semantics ». In : *J. Log. Algebraic Methods Program.* 88 (2017), p. 174-190. doi : 10.1016/j.jlamp.2016.05.001.

[24] Grigore Roşu et Traian Florin Şerbănuţă. « An overview of the K semantic framework ». English (US). In : *Journal of Logic Programming* 79.6 (août 2010), p. 397-434. issn : 1567-8326. doi : 10.1016/j.jlap.2010.03.012.

[25] Amr Sabry et Matthias Felleisen. « Reasoning about Programs in Continuation-Passing Style ». In : *LISP AND SYMBOLIC COMPUTATION*. 1993, p. 288-298.

[26] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar et Rok Strniša. « Ott : Effective Tool Support for the Working Semanticist ». In : *Journal of Functional Programming* 20.1 (jan. 2010), p. 71-122. doi : 10.1017/S0956796809990293.

[27] TC39. *ECMA-262*. url : https://262.ecma-international.org/.

[28] Philip Wadler. « The Essence of Functional Programming ». In : *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992*. Sous la dir. de Ravi Sethi. ACM Press, 1992, p. 1-14. isbn : 0-89791-453-8. doi : 10.1145/143165.143169. url : https://doi.org/10.1145/143165.143169.