Analyse statique de valeurs par interprétation abstraite de programmes fonctionnels manipulant des types algébriques récursifs

Milla Valnet[1], Raphaël Monat[2] et Antoine Miné[3]

[1] École Normale Supérieure, Université PSL, Paris
milla.valnet@ens.psl.eu
[2] Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille
raphael.monat@inria.fr
[3] Sorbonne Université, CNRS, LIP6, F-75005 Paris
antoine.mine@lip6.fr

Résumé

Afin de prévenir les erreurs de programmation, des analyseurs statiques ont été développés pour de nombreux langages ; cependant, aucun analyseur mature ne cible l'analyse de valeurs pour un langage fonctionnel à la ML. Des outils de vérification pour ces langages existent, tels les systèmes de types classiques ou les méthodes déductives, mais le raisonnement automatique sur des programmes numériques a jusqu'alors été peu exploré. Cet article décrit un analyseur statique de valeurs par interprétation abstraite pour un langage fonctionnel typé du premier ordre, approche sûre et automatique pour garantir l'absence d'erreurs à l'exécution. En se basant sur des domaines abstraits relationnels et en réalisant des résumés des champs récursifs des types algébriques, cette approche permet d'analyser des fonctions récursives manipulant des types algébriques récursifs et d'inférer dans un domaine abstrait leur relation entrée-sortie. Une implémentation est en cours sur la plateforme d'analyse multilangage MOPSA et analyse avec succès de courts programmes de quelques lignes. Ce travail ouvre ainsi la voie vers une analyse de valeurs précise et relationnelle basée sur l'interprétation abstraite pour les langages fonctionnels d'ordre supérieur à la ML.

1 Introduction

Même lorsqu'il est fortement et statiquement typé, un programme fonctionnel est toujours susceptible de signaler des erreurs lors de l'exécution — accès à un tableau en dehors de ses bornes, assertions fausses, divisions par zero, échec de filtrage, etc. — ou d'avoir un comportement indésirable tels les dépassements d'entier. Notre objectif est de développer une analyse capable de détecter statiquement de telles erreurs.

L'obtention de garanties sur les langages fonctionnels s'est jusqu'ici très largement reposée sur les systèmes de types, certains trop imprécis pour prouver l'absence d'erreurs, comme l'inférence à la ML, d'autres très précis comme les types de raffinements [31, 32] mais utilisant des solveurs lourds comme boîtes noires. D'autres techniques se basant sur des prouveurs de théorèmes [13, 30] supportent les fonctions récursives sur les types récursifs, mais demandent à l'utilisateur de spécifier les pré- et post-

conditions des fonctions récursives (équivalent des invariants de boucles) pour assister le prouveur. Le choix d'une analyse statique comporte ainsi l'avantage d'une méthode automatique, sans annotation.

Cependant, les analyses statiques de programmes fonctionnels se sont beaucoup concentrées sur l'analyse de flot de contrôle (CFA), [24, 25] un type d'analyse visant à déterminer à quel point du programme une fonction peut être appelée, ou encore sur des analyses de *strictness* ou de terminaison [10]. Ces méthodes ne retiennent aucune information concernant les valeurs des variables numériques et ne permettent donc pas de garantir l'absence de certaines erreurs à l'exécution. Plus récemment, Montagu et al. [23] se sont intéressés aux relations entre les champs des types algébriques et Bautista et al. [2, 3] à celles entre leurs valeurs, mais aucun ne supporte les types récursifs.

Enfin, comparés aux langages impératifs et orientés objets, sur lesquels sont développés la plupart des analyseurs abstraits [4, 6, 12, 29] à ce jour, les langages fonctionnels introduisent des défis nouveaux, rassemblés sur l'exemple suivant :

```
type list = Cons of int * list | Nil
let rec mult2 l =
  match l with
  | Cons(h, q) -> Cons(2*h, mult2 q)
  | Nil -> Nil
in
let hd x = match x with Cons(h,q) -> h | Nil -> assert false in
let x = Cons(0, Cons(1, Cons(2, Nil))) in
assert(hd (mult2 x) <= 4)
```

La fonction `mult2` multiplie par deux tous les éléments de la liste — définie par un type utilisateur — passée en argument, tandis que la fonction `hd` renvoie son premier élément (sa tête). L'objectif de notre analyse est d'être en mesure de prouver l'assertion et donc l'absence d'erreurs à l'exécution. Cela implique d'être capable d'analyser les types algébriques récursifs, les fonctions récursives et le filtrage par motifs. Ce travail fournit donc les contributions suivantes :

- Un domaine abstrait relationnel pour les objets algébriques récursifs, modulaire en le domaine numérique choisi, effectuant des résumés par champ.
- Une analyse de valeurs par interprétation abstraite pour un langage fonctionnel monomorphe du premier ordre capable d'analyser des fonctions récursives manipulant des objets récursifs et d'inférer leur relation entrée-sortie.
- Une implémentation en OCaml dans la plateforme MOPSA, en cours de développement et analysant avec succès de courts programmes écrits en OCaml.

Plan. La section 2 décrit le langage fonctionnel sur lequel nous travaillerons. Les méthodes d'analyse sont présentées et formalisées en section 3. La section 4 s'attarde

sur leur implémentation dans la plateforme MOPSA, tandis que la section 5 détaille l'état de l'art. La section 6 présente les extensions possibles et conclut.

2 Un langage fonctionnel

Dans cette partie, nous décrivons un langage fonctionnel jouet que nous chercherons à analyser dans les parties suivantes.

Alors que les langages impératifs manipulent majoritairement des données numériques, des structures ou encore des pointeurs, les langages fonctionnels utilisent quant à eux largement la notion de *type algébrique* (ou *Algebraic Data Types*, ADTs). Il s'agit d'un type formé par combinaison (sommes ou produits) d'autres types. Nous considérerons ici des types algébriques de la forme suivante :

$$\mathcal{T}_0 ::= int \mid (C_1 \texttt{ of } \tau_{1,1} * \cdots * \tau_{1,m_1} \mid \cdots \mid C_n \texttt{ of } \tau_{n,1} * \cdots * \tau_{n,m_n}) \quad \text{avec } \tau_{i,j} \in \mathcal{T}_0$$
$$\mathcal{T} ::= d \in \mathcal{T}_0 \mid d_1 \to \cdots \to d_n \to d_{n+1}, \quad \text{avec } \forall i, d_i \in \mathcal{T}_0$$

Ainsi, un type est dans $\mathcal{T}_0$ s'il est le type int (type des entiers), ou s'il est un type algébriquement construit à partir de lui-même et des types de $\mathcal{T}_0$, c'est-à-dire sous la forme d'une somme de constructions $C_i \texttt{ of } \tau_{i,1} * \cdots * \tau_{i,m_i}$, où les C_i sont appelés constructeurs de type. Un type t sous cette forme représente ainsi les objets s'écrivant sous la forme $C_i(e_{i,1}, \ldots, e_{i,m_i})$, où $e_{i,j}$ est une expression de type $\tau_{i,j}$.

Ce type peut être vu comme la somme des types $C_i \texttt{ of } \tau_{i,1} * \cdots * \tau_{i,m_i}$, eux-mêmes réalisant le produit de types $\tau_{i,1} * \cdots * \tau_{i,m_i}$. Cette définition des types algébriques, proche de celle de OCaml, n'est pas une définition générale. En effet, une construction unique réunit ici deux constructeurs différents, la somme et le produit, de sorte à obtenir une syntaxe et une sémantique plus compacte. Par souci de simplification, nous ne considérerons pas les types mutuellement récursifs dans ce formalisme, mais tous les raisonnements menés s'y étendraient sans difficulté. De plus, notre analyse se limitera à un langage monomorphe du premier ordre, aussi les types $\tau_{i,j}$ ne peuvent pas être des fonctions, et les types algébriques polymorphes ne sont pas supportés.

L'ensemble $\mathcal{T}$ des types possibles est donc l'ensemble des types $d \in \mathcal{T}_0$, ainsi que le type des fonctions de $d_1, \ldots, d_n$ dans d_{n+1} avec $d_i \in \mathcal{T}_0$. On considérera désormais un programme préfacé par une liste de déclarations de types sous la forme $\texttt{type } t = C_1 \text{ of } \tau_{1,1} * \cdots * \tau_{1,m_1} \mid \cdots \mid C_n \text{ of } \tau_{n,1} * \cdots * \tau_{n,m_n}$. Un type peut être algébriquement construit uniquement à base des types précédemment déclarés, du type int et de lui-même. On note $\mathbb{C}$ l'ensemble (fini) des constructeurs de type C_i utilisés dans le programme. En posant $e_1, \ldots, e_n \in \mathcal{E}, p_1, \ldots, p_n \in \Pi, C \in \mathbb{C}$, nous nous munirons ici du langage « à la ML » dont la syntaxe est détaillée en figure 1.

Ici, $\mathbb{V}$ représente l'ensemble des variables, le symbole $\oplus$ les opérateurs du langage ($+$, $-$, $*$, $/$, $=$, etc.). $fv(e)$ correspond aux variables libres de l'expression e. Π, l'ensemble des motifs (*patterns*), contient ainsi le joker `_`, les variables, les entiers, les clauses `when`, et les constructeurs de types dont les paramètres sont des motifs aux variables libres disjointes. $\bot_X$ symbolise la non-terminaison des expressions à valeur dans X. Les

$$
\begin{aligned}
\mathcal{E} ::= \;& x \in \mathbb{V} \\
& \mid n \in \mathbb{Z} && \mid e_1 \oplus e_2 \\
& \mid e_1 e_2 \ldots e_n && \mid \text{fun } x_1 \ldots x_n \to e \\
& \mid \text{let } x = e_1 \text{ in } e_2 && \mid \text{let rec } x = e_1 \text{ in } e_2 \\
& \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 && \mid \text{assert } e \\
& \mid C(e_1, \ldots, e_n) && \mid (\text{match } e_0 \text{ with } p_1 \to e_1 \mid \cdots \mid p_m \to e_m)
\end{aligned}
$$

$$\Pi ::= \_ \mid \; x \in \mathbb{V} \mid n \in \mathbb{Z} \mid p_1 \text{ when } e_1 \mid C(p_1, \ldots, p_n), \text{ où } \forall i \neq j, fv(p_i) \cap fv(p_j) = \emptyset$$

$$\mathcal{V}_0 ::= \mathbb{Z}^\perp \cup \mathbb{C}(\mathcal{V}_0)^\perp \cup \{\omega\}^\perp, \quad \text{avec } \begin{cases} X^\perp = X \cup \{\perp_X\} \\ \mathbb{C}(X) = \{\; C(x_1, \ldots, x_n) \mid C \in \mathbb{C}, x_i \in X \;\} \end{cases}$$

$$\mathcal{V} ::= \mathcal{V}_0 \cup \Lambda^\perp \quad \text{avec } \Lambda = [\mathcal{V}_0^n \to \mathcal{V}_0]$$

$$\Sigma = \mathbb{V} \to \mathcal{V}$$

Figure 1 – Syntaxe du langage fonctionnel.

valeurs de premier ordre du langage $\mathcal{V}_0$ sont les entiers, les constructeurs de types dont les arguments sont des valeurs de premier ordre et les erreurs ω auxquelles on adjoint $\perp$. L'ensemble des valeurs $\mathcal{V}$ se constitue ainsi de l'ensemble des valeurs du premier ordre ainsi que des fonctions continues des valeurs d'ordre 1 dans les valeurs d'ordre 1 ($\perp$ en cas de non-terminaison). Enfin, Σ représente l'environnement, associant une valeur à chaque variable. La sémantique concrète du langage définie en Annexe A, figure 4.

Nous avons mentionné les types et leur déclaration au début de cette sous-section. Dans la suite, nous travaillerons sous l'hypothèse que nos programmes sont bien typés. L'inférence de type ne sera pas détaillée ici, mais fonctionne de la manière habituelle. Au cours de l'analyse, les informations sur les déclarations de types comme sur les types des variables seront utilisées pour sélectionner le domaine d'abstraction adéquat, en particulier pour les objets algébriques. De plus, cette inférence permet de détecter de manière statique certaines situations dans laquelle une expression s'évalue en ω. Elle échoue cependant à détecter cette erreur lorsque celle-ci est déclenchée par un échec de filtrage (i.e. lorsque l'élément ne correspond à aucun motif), d'assertions ou de division par zéro. Ainsi, c'est sur de telles propriétés, portant sur la valeur des variables, que notre analyse se focalisera.

On notera que le type `int` correspond ici aux entiers mathématiques ; cependant, travailler sur des entiers machines dans le cadre de l'analyse d'OCaml ne représente pas de difficulté supplémentaire majeure. Enfin, le langage défini n'encode pas toutes les caractéristiques des langages fonctionnels. L'ordre supérieur et le polymorphisme ne sont pas supportés, et ce langage est fonctionnel *pur*, ce qui signifie qu'il ne permet aucun effet de bord — ceci exclut en particulier l'utilisation des références et des tableaux. Ce langage contient ainsi les caractéristiques fonctionnelles sur lesquelles nous travaillerons, les fonctions et types algébriques récursifs. Cela permettra en particulier d'analyser les

fonctions récursives manipulant des structures de données récursives.

3 Formalisation des domaines

3.1 L'interprétation abstraite

Le domaine des valeurs et les opérateurs définissant la sémantique concrète du langage de la section 2 sont trop complexes à manipuler : les propriétés intéressantes sont incalculables. On utilise la théorie de l'interprétation abstraite [9], qui consiste à remplacer un domaine concret C par un domaine abstrait plus simple, A, aux éléments représentables en mémoire, munis respectivement d'ordre partiels $\leq$ et $\sqsubseteq$ correspondant au niveau d'information contenu dans les éléments.

Une fonction de *concrétisation* $\gamma : A \to C$ donne alors le sens d'un élément abstrait, en expliquant quel élément concret il peut représenter, et doit être compatible avec l'ordre d'information — croissante, donc. Pour $a \in A, c \in C, \gamma(a)$ est alors la concrétisation de a dans C, et si $\gamma(a) \geq c$, alors a est une approximation correcte de c.

Par exemple, dans un programme manipulant des valeurs numériques, chaque variable peut être représentée par l'ensemble de ses valeurs possibles, dans $\mathcal{P}(\mathbb{Z})$, ordonné par l'inclusion. Cependant, cet ensemble étant de taille non bornée, son stockage et sa manipulation sont coûteux. En choisissant comme ensemble abstrait des intervalles de valeurs [8], on sur-approxime $\mathcal{P}(\mathbb{Z})$ en un ensemble plus léger à manipuler. Par exemple, on peut abstraire $\{1, 3, 18, 37\}$ par $[1, 37]$.

Il s'agit ensuite d'être capable d'abstraire les opérateurs du concret, telle l'addition des entiers, en des opérateurs sur les éléments abstraits. Une *abstraction sûre* de $f : C \to C$ est alors une fonction $f^\sharp : A \to A$ telle que $\forall a \in A, f(\gamma(a)) \leq \gamma(f^\sharp(a))$.

Enfin, pour interpréter des boucles, ou dans notre cas, des fonctions récursives, analyser chaque itération puis effectuer l'union des résultats possibles demande un nombre d'opérations arbitrairement grand, voire infini. On doit donc se munir d'un opérateur capable d'assurer la convergence de telles itérations, l'élargissement (*widening*) $\nabla : A \times A \to A$, qui sur-approxime l'union et force la convergence dans A en temps fini.

À partir de cette théorie, on trouve dans la littérature de nombreux domaines abstraits numériques abstrayant $\mathcal{P}(\mathbb{Z}^n)$, avec différents compromis entre coût et précision (intervalles, octogones, polyèdres, etc.), dont certains sont relationnels — i.e. ils permettent de garder trace des relations entre les variables plutôt que de les abstraire indépendamment. Nous les exploiterons par la suite pour construire nos domaines pour les types algébriques.

3.2 Abstractions relationnelles avec des variables symboliques

Nous présentons ensuite les domaines abstraits utilisés pour définir une sémantique abstraite calculable $\mathbb{E}^\sharp$. Traditionnellement, une telle sémantique prend en paramètre une expression et renvoie une valeur abstraite dans le domaine pertinent. Cependant,

$$\mathcal{E}_{\mathbb{Z}} ::= n \in \mathbb{Z} \mid v \in \mathbb{V}_{\mathbb{Z}} \mid e_1 \oplus e_2, \text{ où } e_1, e_2 \in \mathcal{E}_{\mathbb{Z}}$$
$$\mathcal{V}^\sharp = \mathcal{D} \cup \mathbb{V}_{\mathbb{Z}}$$
$$\mathcal{E}^\sharp = \mathcal{D} \cup \mathcal{E}_{\mathbb{Z}}$$
$$\Sigma^\sharp = (\mathbb{V} \to \mathcal{V}^\sharp) \times \mathcal{D}_{\mathbb{Z}}$$

Figure 2 – Formalisation de la sémantique abstraite.

cette méthode ne garde pas trace des relations entre variables numériques.

```
let x = random 1 10 in
let y = x + 1 in y
```

En effet, une telle analyse interprétera sur le programme ci-contre `x` et `y` comme des valeurs abstraites et pourra inférer, dans le domaine des intervalles, `x -> [1,10]` et `y -> [2,11]`, mais pas l'information relationnelle `y = x + 1`. Pour pallier cela, on choisit d'abstraire une expression numérique par une expression symbolique, qui peut contenir des variables, et un environnement abstrait adjoint à un domaine numérique, qui garde la relation entre ces variables. Ce principe est utilisé dans l'analyseur MOPSA [19], pour l'analyse de C et de Python, et permet aux expressions non-numériques d'exploiter la relationnalité de domaines abstraits tels les domaines numériques relationnel — relation entre tailles de chaînes de caractère, par exemple.

La figure 2 définit nos ensembles abstraits. En notant $\mathbb{V}_{\mathbb{Z}}$ l'ensemble des variables représentant des entiers, $\mathcal{E}_{\mathbb{Z}}$ est l'ensemble des expressions numériques, et contient donc des variables symboliques. On note ensuite $\mathcal{D}$ l'union des domaines des objets algébriques et des fonctions, définis par la suite, et $\mathcal{D}_{\mathbb{Z}}$ un domaine abstrait numérique existant sur $\mathbb{V}_{\mathbb{Z}}$ — tels les intervalles ou les polyèdres. En particulier, il peut être relationnel. $\mathcal{V}^\sharp$ correspond alors à l'ensemble des valeurs abstraites, où les valeurs numériques sont représentées par des variables, et $\mathcal{E}^\sharp$ à l'ensemble des expressions abstraites. On note enfin $\Sigma^\sharp$ l'environnement abstrait : le premier composant est une fonction des variables dans les valeurs abstraites, le second composant donne une valeur aux variables numériques, et notamment aux variables symboliques. On cherche dans cette partie à définir la sémantique abstraite $\mathbb{E}^\sharp[\![.]\!]\ . : \mathcal{E} \times \Sigma^\sharp \to \mathcal{E}^\sharp \times \Sigma^\sharp$.

Enfin, on note $s[x \to v]$ l'association de la valeur $v \in \mathcal{V}^\sharp$ à la variable $x \in \mathbb{V}$ dans un environnement $s \in (\mathbb{V} \to \mathcal{V}^\sharp)$, et avec $\mathbb{E}_{\mathbb{Z}}$ la fonction de transfert, supposée donnée, du domaine numérique, $\mathbb{E}_{\mathbb{Z}}[\![v_x \leftarrow v]\!]$ est l'association de la variable v_x à une valeur ou une expression symbolique v dans le domaine numérique. Elle renvoie un environnement abstrait dont la deuxième composante d est mise à jour. On note $\sqcup_{\Sigma^\sharp}$ l'union sur les environnements. On définit alors l'association de $v \in \mathcal{V}^\sharp$ à $x \in \mathbb{V}$ dans $(s, d) \in \Sigma^\sharp$:

$$\forall (s,d) \in \Sigma^\sharp, (s,d)[x \to v] = \begin{cases} \mathbb{E}_{\mathbb{Z}}[\![v_x \leftarrow v]\!]((s[x \to v_x], d)) & \text{si } x : \texttt{int} \\ (s[x \to v], d) & \text{sinon} \end{cases}$$

Lorsque x est entière, on la représente par la variable v_x dans $\mathbb{V} \to \mathcal{V}^\sharp$, avant d'ef-

fectuer l'assignation de v_x à v dans le domaine numérique. Sinon, on associe x à v dans s. Dans ce contexte, la sémantique abstraite de notre programme peut renvoyer : $(v_y, ([x \to v_x][y \to v_y], [1 \leq v_x \leq 10][v_y = v_x + 1]))$. La variable numérique v_y correspondant à y est alors bien en relation avec la variable numérique v_x correspondant à x. On différencie ici la variable numérique v_x sur la quantité numérique et la variable de programme `x`. Notons enfin que $\forall e \in \mathcal{E}$, $\mathbb{E}^\sharp[\![e]\!]\ \emptyset = (\bot^\sharp, \emptyset)$. Par souci de simplification, on supposera par la suite que les erreurs sont propagées implicitement : si l'abstraction d'une sous-expression contient ω, l'expression aussi.

3.3 Une abstraction pour les types algébriques

Les types algébriques sont fréquemment manipulés en programmation fonctionnelle — on peut citer le type des listes d'entiers simplement chaînées :

```
type list = Cons of int * list | Nil
```

On souhaite pouvoir analyser des programmes manipulant des objets de ce type. Plus généralement, on cherche à disposer d'un domaine spécialisé pour des objets dont le type est un type algébrique (potentiellement récursif) de la forme :

$$\texttt{type}\ \tau = C_1\ \text{of}\ \tau_{1,1} * \cdots * \tau_{1,m_1} \mid \cdots \mid C_n\ \text{of}\ \tau_{n,1} * \cdots * \tau_{n,m_n}$$

Bautista et al. [2, 3] s'intéressaient déjà aux types algébriques lorsque ceux-ci étaient non récursifs. À notre connaissance, aucun domaine visant à abstraire les valeurs prises par un type algébrique *récursif* n'a pour l'heure été créé. On note $\mathbb{C}_\tau = \{C_i \mid 1 \leq \texttt{i} \leq m\}$ l'ensemble des constructeurs du type τ et $\mathcal{T}_\tau$ l'ensemble des objets concrets *finis* de type τ. Remarquons que ces objets sont cependant de taille non bornée, et $\mathcal{T}_\tau$ peut être infini. On cherche à définir, pour le type $\tau_{i,j}$, un domaine abstrait $\mathcal{D}^\sharp_{i,j}$ pour représenter tous les objets de type $\tau_{i,j}$ contenus dans le champ j des constructeurs C_i accessibles dans un objet de type τ.

$$\mathcal{D}^\sharp_{i,j} = \begin{cases} \mathcal{P}(\mathbb{C}_\tau) & \text{si } \tau_{i,j} = \tau \\ \mathbb{V}_\mathbb{Z} & \text{si } \tau_{i,j} = \texttt{int} \\ \mathcal{D}^\sharp_{\tau_{i,j}} & \text{le domaine du type } \tau_{i,j} \text{ déjà défini sinon} \end{cases}$$

On cherche désormais à définir un ensemble abstrait pour l'ensemble concret $\mathcal{T}_\tau$:

$$\mathcal{D}^\sharp_\tau = \prod_{\substack{1 \leq i \leq m \\ 1 \leq j \leq n_i}} (\mathcal{D}^\sharp_{i,j})^\bot \times \mathcal{P}(\mathbb{C}_\tau)$$

Intuitivement, $\mathcal{D}^\sharp_\tau$ abstrait un objet x de type τ en un objet abstrait $(g, \mathscr{C})$, où :

— $g \in \prod (\mathcal{D}^\sharp_{i,j})^\bot$ est tel qu'en notant $g_{i,j}$ l'élément de $(\mathcal{D}^\sharp_{i,j})^\bot$ dans le tuple g, celui-ci abstrait toutes les valeurs du j-ième élément des constructeurs C_i de x.

— $\mathscr{C} \in \mathcal{P}(\mathbb{C}_\tau)$ est l'ensemble des constructeurs C_i possibles pour x

On crée alors $x.i.j$ une variable numérique symbolique, qui représente toujours le champ j des constructeurs C_i accessibles dans x lorsque celui-ci est entier. Ainsi, le domaine numérique peut inférer des relations entre ces variables. Remarquons alors que $\mathcal{D}^\sharp_{i,j} \neq \mathcal{D}^\sharp_{\tau_{i,j}}$ lorsque $\tau_{i,j} = \tau$ ou `int`.

Exemple 3.1. Dans le cas des listes d'entiers, en choisissant pour $\mathcal{D}_{\mathbb{Z}}$ l'ensemble des intervalles $\mathbb{I}$, on a $\mathbb{C}_{\texttt{list}} = \{\texttt{Cons}, \texttt{Nil}\}$, $\mathcal{D}^{\sharp}_{\texttt{list}} = (\mathbb{V}^{\perp}_{\mathbb{Z}} \times \mathcal{P}(\mathbb{C}_{\texttt{list}})^{\perp}) \times \mathcal{P}(\mathbb{C}_{\texttt{list}})$. En effet, le constructeur `Cons` a deux champs : le premier contient des entiers, représenté par les variables numériques $\mathbb{V}_{\mathbb{Z}}$ — associées dans un environnement σ à des éléments de $\mathbb{I}$ — et le second, récursif, des listes, donc représenté par $\mathcal{P}(\mathbb{C}_{\texttt{list}})$. Pour un objet x, $x.1.1$ représente alors le champ 1 du premier constructeur, donc le champ `int` de `Cons`. Un objet de $\mathcal{D}^{\sharp}_{\texttt{list}}$ serait $((r.1.1, \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}\}) \in \mathcal{D}^{\sharp}_{\texttt{list}}$, avec $[r.1.1 \to [1, 10]] \in \sigma$.

L'étape suivante est de définir la concrétisation γ_τ qui à un objet abstrait de $\mathcal{D}^{\sharp}_{\tau}$ dans un environnement abstrait associera un ensemble d'objets concrets de $\mathcal{T}_\tau$ pouvant lui correspondre. On note $\gamma_{\tau_{i,j}}$ la concrétisation associée au domaine de $\tau_{i,j}$ lorsque $\tau_{i,j} \neq \tau$.

Définition 3.1. Pour $(g, \mathscr{C}) \in \mathcal{D}^{\sharp}_{\tau}$, $\sigma^{\sharp}$ un environnement abstrait, on a la concrétisation :

$$\gamma_\tau((g, \mathscr{C}), \sigma^{\sharp}) = \{\ x : \tau \mid x = C_i(x_{i,1}, \ldots, x_{i,n_i}) \ \wedge\ C_i \in \mathscr{C} \ \wedge \\ \forall j, x_{i,j} \in \begin{cases} \gamma_\tau((g, g_{i,j}), \sigma^{\sharp}) & \text{si } \tau_{i,j} = \tau \\ \gamma_{\tau_{i,j}}(g_{i,j}, \sigma^{\sharp}) & \text{sinon} \end{cases} \}$$

Lorsque le type τ est fini, la récursion est bien fondée ($x_{i,j}$ contient strictement moins de constructeurs que x) et cette définition est correcte.

Intuitivement, un objet x de type τ peut alors être abstrait par $(g, \mathscr{C})$ si :

— Il s'écrit à partir d'un constructeur C_i de $\mathscr{C}$.

— Chaque j-ième champ de constructeur C_i accessible dans x a :

- son constructeur dans $g_{i,j}$, quand il est récursif ($\tau_{i,j} = \tau$) ;
- ou peut être abstrait par $g_{i,j}$ s'il est non récursif.

Ainsi, $g_{i,j}$ représente donc un résumé de tous les champs j de constructeur C_i accessibles dans x. L'environnement en paramètre permet de garder trace des variables numériques.

Un objet d'un type récursif étant de taille non bornée, il s'agit ici d'obtenir un domaine abstrait permettant de le représenter de manière finie. Pour ce faire, on « replie » chaque champ de constructeurs en un résumé, de sorte à ne mémoriser qu'une variable *résumé* pour chacun d'eux. Cette approche comporte des similitudes avec l'idée de Gopan et al. [14] qui proposait d'utiliser un nombre fixé de dimensions pour sur-approximer les valeurs d'une collection non bornée d'objets numériques, dans le cadre d'analyses de tableaux dynamiques ou de structures allouées sur le tas. Remarquons que le « résumé » ou « repliement » est effectué au site d'allocation de la variable de type algébrique — l'abstraction est dite *object-sensitive* [28].

Exemple 3.2. Dans le cas des listes, en notant $(g, \mathscr{C}) = ((r, \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}\})$ et $\sigma^{\sharp} = [r \to [1, 10]]$, la concrétisation $\gamma_{\texttt{list}}((g, \mathscr{C}), \sigma^{\sharp})$ vaut :

$$\{\ x : \texttt{list} \mid x = \texttt{Cons}(h, q), q \in \gamma_{\texttt{list}}(((r, \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}, \texttt{Nil}\}), \sigma^{\sharp}) \wedge h \in [1, 10]\ \}$$

En particulier, $\texttt{Cons}(1, \texttt{Cons}(4, \texttt{Cons}(10, \texttt{Nil}))) \in \gamma_{\texttt{list}}((([1, 10], \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}\}), \sigma^{\sharp})$, ce qui signifie que $((r, \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}\})$ est une sur-approximation correcte de

$\texttt{Cons}(1, \texttt{Cons}(4, \texttt{Cons}(10, \texttt{Nil})))$ dans $\sigma^\sharp$. De manière générale, c'est une sur-approximation correcte pour toutes les listes non vides dont les éléments sont dans $[1, 10]$.

Pour définir un domaine abstrait, on doit également se munir d'une relation d'ordre entre les éléments de $\mathcal{D}^\sharp_{\mathtt{t}}$.

Définition 3.2. On note $\sqsubseteq^\perp_{i,j}$ l'ordre plat sur $\mathcal{D}^{\sharp,\perp}_{i,j}$. On définit ensuite l'opérateur $\sqsubseteq$:

$$(g^1, \mathscr{C}_1) \sqsubseteq (g^2, \mathscr{C}_2) \iff \mathscr{C}_1 \subseteq \mathscr{C}_2 \wedge (\forall 1 \leq i \leq n, C_i \in \mathscr{C}_1 \implies \forall 1 \leq j \leq m_i, g^1_{i,j} \sqsubseteq^\perp_{i,j} g^2_{i,j})$$

Exemple 3.3. Pour les listes, $((x.1.1, \mathscr{C}_1), \mathscr{C}'_1) \sqsubseteq ((y.1.1, \mathscr{C}_2), \mathscr{C}'_2) \iff \mathscr{C}'_1 \subseteq \mathscr{C}'_2 \wedge \mathscr{C}_1 \subseteq \mathscr{C}_2 \wedge x.1.1 \sqsubseteq_{\mathbb{I}} y.1.1$. Par exemple, $((x.1.1, \{\texttt{Nil}\}), \{\texttt{Cons}\}) \sqsubseteq ((y.1.1, \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}\})$ si $x.1.1 \to [1, 4]$ et $y.1.1 \to [1, 10]$. $x.1.1$ et $y.1.1$ représentent les valeurs numériques dans les listes d'entiers x et y.

On construit maintenant une abstraction $\cup^\sharp_\tau$ sûre pour $\cup$:

$$(g^1, \mathscr{C}_1) \ \cup^\sharp_\tau \ (g^2, \mathscr{C}_2) = (g, \mathscr{C}_1 \cup \mathscr{C}_2) \quad \text{avec } \forall i \leq n, \forall j \leq m_j, g_{i,j} = g^1_{i,j} \ \cup_{\mathcal{D}_{i,j}} g^2_{i,j}$$

On définit l'élargissement de la même manière, en remplaçant $\cup^\sharp_\tau$ et $\cup_{\mathcal{D}_{i,j}}$ par ∇_τ et $\nabla_{\mathcal{D}_{i,j}}$. On définit de même l'intersection abstraite. Les preuves de sûreté sont fournies en annexe D.1 et D.2. Il s'agit ensuite de définir les fonctions de transfert pour la construction d'objets de type algébrique, c'est-à-dire, étant donné un environnement abstrait $\sigma^\sharp$ et une expression concrète $C_k(e_1, \ldots, e_n)$, en calculer une abstraction $(g, \mathscr{C})$ dans un nouvel environnement abstrait. Pour ce faire, on utilise un label syntaxique frais r pour l'expression et les $r.i.j$ sont alors des variables fraîches résumant le champ entier du j-ème élément du constructeur i, dont la valeur $h_{i,j}$ est stockée dans l'environnement.

$$\mathbb{E}^\sharp[\![\ C_k(e_1, \ldots, e_n)^r \ ]\!]\sigma^\sharp = (g, \{C_k\}), \sigma^\sharp_0[\forall i \leq n, j \leq m_i, \ r.i.j \to h_{i,j}] \quad \text{avec}$$

$$\begin{cases} v_i, \sigma^\sharp_i = \mathbb{E}^\sharp[\![e_i]\!]\sigma^\sharp \\ (g^0, \mathscr{C}) = \bigcup^\sharp_{e_i : \tau} v_i \\ \sigma^\sharp_0 = \bigsqcup^{\Sigma^\sharp}_i \sigma^\sharp_i \end{cases} \qquad h_{i,j} = \begin{cases} g^0_{k,j} \cup_{\mathcal{D}_{k,j}} v_j & \text{si } \begin{array}{l} i = k \\ \tau_{k,j} \neq \tau \end{array} \\ g^0_{i,j} & \text{sinon} \end{cases} \qquad g_{i,j} = \begin{cases} r.i.j & \text{si } \tau_{i,j} = \texttt{int} \\ h_{i,j} & \text{sinon} \end{cases}$$

L'idée est la suivante : la sémantique abstraite de $C_k(e_1, \ldots, e_n)$ est un objet x du domaine $\mathcal{D}^\sharp_\tau$. Le seul constructeur possible pour cet objet est C_k. On abstrait ensuite ses champs de la manière suivante, dans la variable g : on note $(g^0, \mathscr{C})$ l'union abstraite des évaluations de tous ses champs récursifs — les e_i de type τ. De la sorte, le champ $g^0_{i,j}$ représente l'union de tous les contenus des champs j des C_i accessibles depuis l'objet x.

On définit g en définissant ses composantes $g_{i,j}$, censées représenter tous les objets dans un champ j de constructeur C_i accessibles dans x :

— Si $g_{k,j}$ correspond à un champ non récursif de C_k : il vaut l'union de tous les champs j des C_k accessibles, c'est-à-dire l'union de la sémantique abstraite de e_j — un champ j de C_k — avec $g^0_{k,j}$ — résumé de tous les autres champs j des C_k.

— Sinon : tous les objets dans un champ j de constructeur C_i sont accessibles uniquement via les champs récursifs, et donc résumés dans $g^0_{i,j}$.

Les champs $g_{i,j}$ de type `int` sont alors effectivement assignés à la variable $r.i.j$, de valeur $h_{i,j}$ dans l'environnement. Cette indirection permet, dans le cas où $\mathcal{D}_{\mathbb{Z}}$ est relationnel, d'obtenir des relations entre les $r.i.j$, comme illustré dans l'exemple 3.6. On remarque ainsi que ce domaine s'appuie largement sur les domaines numériques. Sa preuve de sûreté est fournie en annexe D.3.

Exemple 3.4. Ainsi, on peut écrire

$$\begin{aligned}
\mathbb{E}^\sharp[\![\texttt{Nil}]\!]\sigma^\sharp &= ((r.1.1, \bot), \{\texttt{Nil}\}), \sigma^\sharp[r.1.1 \to \bot] \\
\mathbb{E}^\sharp[\![\texttt{Cons}(10, \texttt{Nil})]\!]\sigma^\sharp &= ((r.1.1, \{\texttt{Nil}\}), \{\texttt{Cons}\}), \sigma^\sharp[r.1.1 \to \bot \cup_{\mathbb{I}} [10, 10]] \\
\mathbb{E}^\sharp[\![\texttt{Cons}(1, \texttt{Cons}(10, \texttt{Nil}))]\!]\sigma^\sharp &= (r.1.1, \{\texttt{Nil}\} \cup \{\texttt{Cons}\}), \{\texttt{Cons}\}), \sigma^\sharp[x \to [1, 10]]
\end{aligned}$$

On remarque ici qu'on utilise largement les informations de type du programme. En effet, le type des champs i, j (`t` ou `int`, par exemple) guide l'abstraction en renvoyant la gestion des champs vers les domaines pertinents ; pour ce faire, il est nécessaire de connaître la définition du type `t`. On repose donc très fortement sur l'hypothèse que le programme analysé est bien typé.

Exemple 3.5. On peut illustrer l'utilisation de ce domaine abstrait :

```
type tree = Node of tree * int * tree | Leaf of int
let x = Node(Node(Leaf(250), 100, Leaf(251)), 1, Leaf(252))
```

Dans ce contexte, $\mathbb{E}^\sharp[\![\texttt{Node}(\texttt{Node}, (\texttt{Leaf}(250), 100, \texttt{Leaf}(251)), 1, \texttt{Leaf}(252))]\!][\,]$ vaut donc $(g, \mathscr{C}) = ((\{\texttt{Node}, \texttt{Leaf}\}, x.1.2, \{\texttt{Leaf}\}, x.2.1), \{\texttt{Node}\})$ dans l'environnement $\sigma^\sharp = ([v_{x.1.2} \to x.1.2][v_{x.2.1} \to x.2.1], [v_{x.1.2} \to [1, 100]][v_{x.2.1} \to [250, 252]])$.

$$\begin{aligned}
\gamma((g, \mathscr{C}), \sigma^\sharp) = \{x \mid x = \texttt{Node}(t_1, n, t_2) \ \wedge \ n \in [1, 100] \wedge\ & t_1 \in \gamma((g, \{\texttt{Node}, \texttt{Leaf}\}) \\
& t_2 \in \gamma((g, \{\texttt{Leaf}\})\}
\end{aligned}$$

Cela correspond à l'ensemble des arbres construits à partir de `Node` et poussant uniquement vers la gauche, dont les entiers des constructeurs `Leaf` sont entre 250 et 252 et ceux de `Node` entre 1 et 100.

Cet exemple permet de constater qu'une telle analyse perd beaucoup en précision dans des programmes simples. On pourrait par exemple souhaiter pouvoir garder trace du contenu exact de `x` dans un tel exemple, et ne pas effectuer de résumé des champs. Une proposition pour améliorer la précision de l'analyse décrite ci-dessus serait de n'effectuer un résumé d'un champ que lorsque nécessaire pour la convergence, i.e. lors d'une boucle ou d'un appel récursif, ou encore uniquement à partir d'une certaine profondeur n de l'objet. De telles améliorations n'ont pas encore été implémentées.

Cet exemple permet également de voir que bien qu'ils soient de mêmes types, le deuxième champ du constructeur `Node` est résumé dans une variable différente que le champ du constructeur `Leaf`. Cela permet de gagner en précision, notamment lorsque ces champs revêtent des significations différentes.

Exemple 3.6. L'analyse est également capable d'inférer des propriétés relationnelles :

```
let z = Cons(a, Nil) in
let x = if y <= 2*a + 1 then Cons(y, z) else Cons(2 * a + 1, z)
```

En choisissant comme domaine numérique le domaine des polyèdres [7], on est ici en mesure de prouver : $x_{1,1} \leq 2a + 1$. Cela signifie que toutes les éléments de la liste sont inférieurs à $2a + 1$. Cet aspect relationnel est crucial dans l'inférence des relations entrées-sorties des fonctions (section 3.5), effectuée sans information sur les arguments.

3.4 Abstraction du filtrage par motifs

On s'intéressera ici à la fonction de transfert du filtrage par motifs. Celles des autres constructions du langage sont très standard et sont détaillées en Annexe B. Pour ce faire, on se munit de la fonction abstraite $\texttt{match}^\sharp : \mathcal{V}^\sharp \times \Sigma^\sharp \times \Pi \to \Sigma^\sharp \times \Sigma^\sharp$, détaillée en Annexe C.3. Elle prend en paramètre une valeur abstraite e, un environnement abstrait $\sigma^\sharp$, et un motif p, et renvoie un environnement abstrait restreint dans lequel p et e correspondent, et un dans lequel ils ne correspondent pas. On utilise pour ce faire la fonction de filtre $\mathcal{F}^\sharp$, fournie par le domaine numérique : étant donnée une expression numérique et un environnement abstrait, elle restreint cet environnement abstrait de sorte que l'expression numérique en paramètre soit non nulle. Par exemple, cela permet d'écrire lorsque le motif s'écrit p when e_1 :

$$\texttt{match}^\sharp(\sigma^\sharp, v^\sharp, p \text{ when } e_1) = \mathcal{F}^\sharp[\![e_1^n = 0]\!]\sigma_1^\sharp, \sigma_{\neg,0}^\sharp \text{ avec } \begin{cases} \texttt{match}^\sharp(\sigma^\sharp, v^\sharp, p) = \sigma_0^\sharp, \sigma_{\neg,0}^\sharp \\ \mathbb{E}^\sharp[\![e_1]\!]\sigma_0^\sharp = e_1^n, \sigma_1^\sharp \end{cases}$$

Ainsi, $v^\sharp$ et p when e_1 correspondent dans l'environnement où $v^\sharp$ et p correspondent, restreint au cas où e_1^n est non nul.

$$\mathbb{E}^\sharp[\![\text{match } e_0 \text{ with } p_1 \to e_1 \mid \cdots \mid p_m \to e_m]\!]\sigma^\sharp = \begin{cases} \mathbb{E}^\sharp[\![e_1]\!]\sigma_1^\sharp \cup^\sharp \mathbb{E}^\sharp[\![e_1']\!]\sigma_{\neg,1}^\sharp & \text{si } m \geq 1 \\ \{\omega\}, \sigma^\sharp & \text{si } m = 0 \end{cases}$$

$$\text{avec } \begin{cases} \sigma_1^\sharp, \sigma_{\neg,1}^\sharp = \texttt{match}^\sharp(\mathbb{E}^\sharp[\![e_0]\!]\sigma^\sharp, p_1) \\ e_1' = \text{match } e_0 \text{ with } p_2 \to e_2 \mid \cdots \mid p_m \to e_m \end{cases}$$

L'interprétation du filtrage de motif est l'union abstraite des interprétations e_i dans un environnement dans lequel e_i correspond au motif p_i mais pas aux motifs p_j avec $j < i$, $\{\omega\}$ si e_0 ne correspond à aucun motif. Cela correspond à l'intuition que l'expression e_0

est comparée successivement aux différents motifs, jusqu'à la première correspondance. On remarque qu'on renvoie une erreur lorsqu'après lecture du dernier motif, on analyse un filtrage sans motif dans un environnement non vide : cela correspond au cas où il reste des environnements possibles n'ayant été filtrés par aucun motif. Ainsi, notre analyse cible les échecs de filtrage, d'assertions, de division par zero et de dépassements d'entier — si le domaine numérique les traite. Les autres erreurs sont déjà évitées par le typage. La preuve de sûreté pour ces fonctions de transfert est fournie en Annexe D.3.

Exemple 3.7. Sur un exemple manipulant la structure de filtrage de motifs :

```
let x = match Cons(1, Nil) with
  | Cons(h,q) -> h
  | Nil -> 0
 in assert (x = 1)
```

Informellement, on obtient comme abstraction $\mathbb{E}^\sharp[\![\texttt{Cons}(h,q)]\!]\sigma_1^\sharp \cup^\sharp \mathbb{E}^\sharp[\![\texttt{Nil}]\!]\sigma_{\neg,1}^\sharp$ avec $\sigma_1^\sharp = [h \to 1]$ et $\sigma_{\neg,1}^\sharp = \emptyset$. Le résultat sera donc $h \cup^\sharp \bot^\sharp = h$ dans l'environnement $[h \to 1]$. Ainsi, on est en mesure d'inférer que `x = 1` et l'assertion est donc prouvée correcte.

3.5 Fonctions récursives

Les fonctions récursives sont un incontournable des langages à la ML tel OCaml. Nous avons choisi ici de représenter les fonctions du premier ordre par des relations entre les variables d'entrées et la sortie, en se reposant donc sur un domaine relationnel pour inférer les relations d'entrée-sortie. Le langage étant pur, celles-ci correspondent simplement à l'état relationnel à la fin de la fonction, réduit aux arguments et à la valeur de retour. On s'intéresse aux fonctions de type $\tau = \tau_1 \to \cdots \to \tau_{n+1}$ avec $\tau_1, \cdots, \tau_{n+1} \in \mathcal{T}_0$. On note $\mathcal{D}_i^\sharp$ le domaine choisi pour représenter τ_i.

Définition 3.3. On note $\mathcal{D}_\mathfrak{t}^f = \mathcal{P}(\mathbb{V}) \times \mathcal{E}^\sharp \times \Sigma^\sharp$ le domaine choisi pour représenter les fonctions de type τ. On considère alors la concrétisation :

$$\gamma_f(x_1,\ldots,x_n,e^\sharp,\sigma^\sharp) = \{\ f : \tau \mid \forall v_1:\tau_1,\ldots,v_n:\tau_n, \forall a_1 \in \mathcal{D}_1^\sharp,\ldots,a_n \in \mathcal{D}_n^\sharp, \forall i,$$
$$v_i \in \gamma(a_i,\sigma^\sharp) \wedge f(v_1,\ldots,v_n) \in \gamma(e^\sharp,\sigma^\sharp[x_1=a_1,\ldots,x_n=a_n])\}$$

Ainsi, une fonction est abstraite par : le nom $x_1,\ldots,x_n$ de tous ses arguments formels, la valeur abstraite de son résultat $e^\sharp$, et l'environnement $\sigma^\sharp$ dans lequel ce résultat est valide. Lorsqu'elle est appliquée à des arguments réels de valeurs v_i abstraites par a_i, on ajoute dans l'environnement les contraintes d'égalité entre les arguments x_i et les v_i. On définit ensuite les fonctions de transfert :

$$\mathbb{E}^\sharp[\![\text{fun } x_1 \cdots x_n \to e_1]\!]\sigma^\sharp = (x_1,\ldots,x_n,\mathbb{E}^\sharp[\![e_1]\!]\sigma^\sharp[x_1 \to \top,\ldots,x_n \to \top]),\sigma^\sharp$$

$$\mathbb{E}^\sharp[\![\text{let rec } f = e_1 \text{ in } e]\!]\sigma^\sharp = \mathbb{E}^\sharp[\![e]\!]\sigma^\sharp[f \to \nabla_\tau \underset{n\in\mathbb{N}}{F^n}((x_1,\ldots,x_k,\bot,\sigma^\sharp[\forall i \le k, x_i \to \top]))]$$

$$\text{avec } F(v^\sharp) = fst(\mathbb{E}^\sharp[\![e_1]\!]\sigma^\sharp[f \to v^\sharp]),\ f:\tau,\ \tau = \tau_1 \to \ldots \to \tau_k \to \tau_{k+1}$$

$$\mathbb{E}^\sharp[\![e_0 \ldots e_n]\!]\sigma^\sharp = e^\sharp,\sigma_n^\sharp[x_1 = e_1^\sharp,\ldots,x_n = e_n^\sharp], \quad \text{avec } \begin{cases} \mathbb{E}^\sharp[\![e_0]\!]\sigma^\sharp = (x_1,\ldots,x_n,e^\sharp,\sigma_0^\sharp), \\ \mathbb{E}^\sharp[\![e_{i+1}]\!]\sigma_i^\sharp = e_{i+1}^\sharp,\sigma_{i+1}^\sharp \end{cases}$$

Pour les fonctions définies récursivement, l'intuition est la suivante. On souhaite réaliser une analyse modulaire de la fonction. Pour analyser le corps de la fonction, on considère donc les valeurs des arguments initialisées à $\top$, le maximum du domaine. En effet, pour obtenir la relation entrée-sortie la plus générale possible, on doit supposer les arguments quelconques. De la sorte, on peut appliquer le résumé obtenu à chaque appel, sans contrainte sur la valeur des arguments. Lorsque la fonction est récursive, on cherche à sur-approximer le plus petit point fixe de la sémantique concrète. Par le théorème de Kleene, calculer ce point fixe correspond à itérer l'analyse en débutant à $f \to \bot$. Dans l'abstrait, on procède de même, mais en réalisant un élargissement, ou *widening*, noté ∇_τ, pour assurer la convergence. Enfin, l'application $e_0 \dots e_n$ de la fonction e_0 aux arguments $e_1, \dots, e_n$ ajoute les contraintes d'égalités entre les arguments x_i par l'interprétation abstraite des arguments e_i, dans l'environnement du résultat de la fonction enrichie par les informations obtenues par abstraction des e_j précédents.

On choisit ici d'effectuer l'analyse d'une fonction dès sa déclaration. En effet, cela permet de ne l'analyser qu'une fois, et d'en instancier le résultat à chaque site d'appel selon les valeurs des paramètres. Cela permet une analyse plus efficace. Dans le cas de fonctions mutuellement récursives, non mentionnées ici, notre raisonnement s'étend sans difficulté en itérant sur un vecteur d'abstractions de fonctions.

En notant $\mathbb{A}$ l'ensemble des types algébriques, $\mathcal{D}$ l'union des domaines d'objets algébriques et des domaines de fonctions est alors $\mathcal{D} = \bigcup_{\tau \in \mathbb{A}} \mathcal{D}^\sharp_\tau \cup \bigcup_{\tau=\tau_1 \to \dots \to \tau_n} \mathcal{D}^f_\tau$. Pour obtenir notre domaine abstrait global, on définira l'union $\cup^\sharp$ sur $\mathcal{E}^\sharp \times \Sigma^\sharp$ en Annexe C.2 et de la concrétisation $\gamma : \mathcal{E}^\sharp \times \Sigma^\sharp \to \mathcal{V}$. Sous certaines conditions, il existe une correspondance de Galois (annexe E).

Exemple 3.8. Cependant, cette méthode peut induire une perte de précision :

```
let rec binary = fun x -> if x > 0 then 10 else -10 in binary 1
```

Ici, notre analyse est en mesure de déduire, en analysant `binary` à sa déclaration, qu'étant donnée une entrée quelconque, elle renvoie une valeur dans l'intervalle [-10, 10]. Ainsi, on sait que la valeur de `binary 1` est entre -10 et 10. On aurait pu obtenir un résultat plus précis en analysant le corps de `binary` sachant que l'argument `x` valait 1, ou qu'il était strictement supérieur à 0.

On pourrait également choisir de ne pas être modulaire et d'effectuer cette analyse pour des arguments connus. Ainsi, il suffit de les initialiser non pas à $\top$ mais à leur valeur abstraite. On réalise une analyse identique par élargissement, en se donnant cette fois la possibilité de réaliser des élargissements sur les arguments si les appels récursifs élargissent leur domaine. On pourrait même adapter la méthode de partitionnement, développée par Bourdoncle [5] pour un langage récursif impératif : la fonction est analysée pour plusieurs valeurs abstraites couvrant les arguments d'entrée. Cette méthode permet de gagner en précision sans pour autant analyser ou spécialiser la fonction à chaque appel. Cette implémentation serait l'objet d'un travail futur.

Enfin, l'hypothèse d'un fragment pur est essentielle ici. En effet, une fonction est abstraite comme une relation entre les variables d'entrée et la variable de sortie : cette abstraction est permise puisque la fonction ne réalise aucune modification de l'état mémoire. Dans un fragment impur, contenant tableaux, références ou champs mutables, il serait nécessaire de modifier cette abstraction de sorte à garder trace des modifications de l'état mémoire. Des pistes pour travaux futurs seront résumées en section 6.

3.6 Exemple d'application

Détaillons l'analyse en calculant la sémantique abstraite de l'exemple introductif :

```
let rec mult2 = fun l -> match l with
  | Cons(h, q) -> Cons(2*h, mult2 q)| Nil -> Nil in
let hd = fun y -> match y with Cons(h,q) -> h | Nil -> 0 in
let x = Cons(0, Cons(1, Cons(2, Nil))) in assert(hd (mult2 x) <= 4)
```

On note $body = \text{match } l \text{ with } | \texttt{Cons}(h, q) \to \texttt{Cons}(2 * h, mult2\ q) \mid \texttt{Nil} \to \texttt{Nil}$. On calcule alors $\nabla_{n\in\mathbb{N}} F^n((l, \bot, \sigma^\sharp))$, où $\sigma^\sharp = [l \to \top]$ puisque l est quelconque. Enfin, on note r l'identifiant unique pour le résultat de $body$. Comme l'analyse est générique en le domaine $\mathbb{D}_\mathbb{Z}$ choisi pour représenter les entiers, les opérateurs $+, -, *$ le sont aussi.

$$\begin{aligned}
F((l, \bot, \sigma^\sharp)) &= \mathbb{E}^\sharp[\![\text{fun } l \to body]\!]\sigma^\sharp[mult2 \to (l, \bot, \sigma^\sharp)] \\
&= (l, ((r.1.1, \bot), \{\texttt{Cons}\}, \sigma^\sharp[r.1.1 \to 2 * l.1.1]) \\
&\quad \cup^\sharp_\tau ((r.1.1, \bot), \{\texttt{Nil}\}), \sigma^\sharp[r.1.1 \to \bot]) \\
&= (l, (r.1.1, \bot), \{\texttt{Cons}, \texttt{Nil}\}, \sigma^\sharp[r.1.1 \to 2 * l.1.1]) \\
F^2((l, \bot, \sigma^\sharp)) &= F((l, (r.1.1, \bot), \{\texttt{Cons}, \texttt{Nil}\}, \sigma^\sharp[r.1.1 \to 2 * l.1.1])) \\
&= (l, (r.1.1, \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}, \texttt{Nil}\}), \sigma^\sharp[r.1.1 \to 2 * l.1.1]) \\
F^3((l, \bot, \sigma^\sharp)) &= F^2((l, \bot, \sigma^\sharp)) \\
\nabla_{n\in\mathbb{N}} F^n((l, \bot, \sigma^\sharp)) &= (l, (r.1.1, \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}, \texttt{Nil}\}), \sigma^\sharp[r.1.1 \to 2 * l.1.1]) = V_1
\end{aligned}$$

Par raffinement successif du résumé de la fonction, on est capable d'inférer que la variable résumé des éléments de la liste de retour ($r.1.1$) vaut deux fois la variable résumé des éléments de la liste d'entrée ($l.1.1$). Ainsi, si $\mathcal{D}_\mathbb{Z}$ est le domaine des intervalles et que les éléments de la liste d'entrée sont dans $[a, b]$, alors ceux de la liste de retour sont dans $[2a, 2b]$. On associe ensuite ce résultat à $mult2$ et on poursuit l'analyse de e :

$$\mathbb{E}^\sharp[\![\text{let rec } mult2 = body \text{ in } e]\!]\sigma^\sharp = \mathbb{E}^\sharp[\![e]\!]\sigma^\sharp\sigma^\sharp_1 \quad \text{avec } \sigma^\sharp_1 = \sigma^\sharp[mult2 \to V_1]$$

Puis on évalue $body_2 = \text{match } x \text{ with } | \texttt{Cons}(h, q) \to h \mid \texttt{Nil} \to 0$, qu'on associe à hd.

$$\begin{aligned}\mathbb{E}^\sharp[\![\text{fun } y \to body_2]\!]\sigma_1^\sharp &= (y, \mathbb{E}^\sharp[\![body_2]\!]\sigma^\sharp[y \to \top]), \sigma_1^\sharp \\ &= (y, z, \sigma^\sharp[z \to y.1.1 \cup_{\mathbb{Z}} 0]), \sigma_1^\sharp = V_2, \_ \\ \mathbb{E}^\sharp[\![\text{let } hd = body_2 \text{ in } e_1]\!]\sigma_1^\sharp &= V_2, \sigma_2^\sharp \quad \text{où } \sigma_2^\sharp = \sigma_1^\sharp[hd \to V_2]\end{aligned}$$

Ainsi, hd renvoie z, dont la valeur est le résumé du premier champ de `Cons` pour l ($l.1.1$) joint à 0. Enfin, on calcule l'abstraction de x.

$$\begin{aligned}\mathbb{E}^\sharp[\![\texttt{Cons}(0, \texttt{Cons}(1, \texttt{Cons}(2, \texttt{Nil})))]\!]\sigma_2^\sharp &= ((r_2.1.1, \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}\}), \sigma_3^\sharp) \\ &= V_3, \sigma_3^\sharp \quad \text{où } \sigma_3^\sharp = \sigma_2^\sharp[r_2.1.1 \to 0 \cup_{\mathbb{Z}} 1 \cup_{\mathbb{Z}} 2])\end{aligned}$$

$\mathbb{E}^\sharp[\![\text{let } x = \texttt{Cons}(0, \texttt{Cons}(1, \texttt{Cons}(2, \texttt{Nil}))) \text{ in } e_2]\!]\sigma_2^\sharp = \mathbb{E}^\sharp[\![e_2]\!]\sigma_2^\sharp, \sigma_4^\sharp \quad \text{où } \sigma_4^\sharp = \sigma_3^\sharp[x \to V_3]$

Puis on analyse l'assertion en applicant successivement $mult2$ et hd à x :

$$\begin{aligned}\mathbb{E}^\sharp[\![mult2\ x]\!]\sigma_4^\sharp &= V_1, \sigma_4^\sharp[l \to \sigma_4^\sharp(x)] \\ \mathbb{E}^\sharp[\![hd\ (mult2\ x)]\!]\sigma_4^\sharp &= z_1, \sigma_4^\sharp[l \to \sigma_4^\sharp(x)][y \to V_1][z_1 \to z]\end{aligned}$$

z est la variable résultat de $body_2$, et z_1 est l'interprétation abstraite de l'expression testée par l'assertion. Avec le domaine des polyèdres, on a alors :

$$0 \leq x.1.1 \leq 2 \wedge l.1.1 = x.1.1 \wedge V_1.1.1 \leq 2{*}l.1.1 \wedge\ y.1.1 = V_1.1.1 \wedge 0 \leq z \leq y.1.1 \wedge z_1 = z$$

On déduit $0 \leq V_1.1.1 \leq 4$, puis $0 \leq z_1 \leq 4$. Ainsi, le programme est prouvé correct.

4 Implémentation et expérimentation

4.1 La plateforme MOPSA

L'implémentation s'est effectuée dans MOPSA (Modular Open Platform for Static Analysis) [19], une plateforme en logiciel libre et multi-langage visant à faciliter le développement d'analyseurs statiques par interprétation abstraite.

Elle se différencie des analyseurs statiques existants, tels Astrée [4], Frama-C [6], Infer [12] ou Julia [29] en ce qu'elle tente de combiner des analyses relationnelles basées sur la coopération de domaines abstraits indépendants les uns des autres avec une plateforme modulaire non spécifique à un langage. Elle implémente des analyses ciblant du code écrit en C, en Python, ou mêlant du C et du Python. Elle vérifie l'absence d'erreurs à l'exécution et d'exceptions non rattrapées, et possède un langage de modélisation pour les bibliothèques C, appliqué, par exemple, à la bibliothèque standard. Pour améliorer la précision, les domaines communiquent et coopèrent via produit réduit, composition ou réécriture, ce qui est facilité par une signature commune.

La plateforme est développée en OCaml ; l'environnement de travail consiste en 19 000 lignes de code. Le support pour OCaml, en cours de développement, correspond à

Programme	Lignes	Temps (ms)
`list.ml`	3	3
`tree.ml`	2	5
`match.ml`	4	4
`match_alarm.ml`	4	5
`match_error.ml`	4	4
`add.ml`	3	4

Figure 3 – Temps d'exécution de l'analyse de quelques programmes jouets.

quelques 2000 lignes de code [1]. Celui-ci récupère l'arbre de syntaxe abstraite typé dans le fichier `.cmt` résultant de la compilation d'OCaml, de sorte à bénéficier des parseurs et typeurs natifs et à faciliter la transition vers un support complet du langage. Il supporte actuellement les filtrages de motifs, les types algébriques récursifs et les fonctions non-récursives. Le domaine des fonctions récursives est encore en cours d'implémentation.

4.2 Résultats expérimentaux

Une évaluation expérimentale a été réalisée sur des programmes OCaml écrits à la main. Dans la figure 3, on s'intéresse au temps d'exécution de l'analyse sur un ordinateur portable standard pour de courts programmes jouets déclarant des listes ou des arbres et les manipulant via filtrage de motifs, et dont le code peut être trouvé en Annexe F. Sur des exemples simples, l'analyse trouve les résultats attendus de manière rapide. Pour l'heure, nous nous sommes limités à des programmes jouets de quelques lignes. Une extension serait de s'intéresser à des programmes à taille réelle, provenant de projets ou de la bibliothèque standard.

5 État de l'art

Pour établir des propriétés sur des programmes fonctionnels, les systèmes de types ont été largement utilisés. Les plus simples permettent de prévenir un certain nombre d'erreurs, telle l'addition d'un entier à une fonction. D'autres, tels les types de raffinements [31, 32], permettent d'exprimer des propriétés complexes sur des valeurs algébriques ou numériques mais délèguent pour cela des vérifications à des solveurs SMTs. Les méthodes dites déductives, telles Why3 [13] ou F* [30], peuvent quant à elles établir des propriétés plus fines sur les programmes, telle leur correction par rapport à une spécification. Elles reposent cependant sur des annotations utilisateurs et des solveurs SMT. Notre travail cherche un nouveau compromis, une méthode automatique et sans solveur paramétrée par un choix d'abstraction, pour inférer des propriétés numériques.

Si de nombreuses recherches traitent de l'analyse statique des langages fonctionnels, elles se sont beaucoup concentrées sur l'analyse de flots de contrôles (CFA) [24], ce qui, comme précisé dans [21] ne permet pas de garder trace des valeurs traversant les flots.

1. disponible sur gitlab au lien https://gitlab.com/mvalnet/mopsa-analyzer/-/tree/ocaml

Cousot et Cousot [10], eux, se concentrent sur les fonctions comme objets de première classe mais ont pour but d'exprimer par interprétation abstraite des analyses classiques telles la *strictness* ou la terminaison et ne proposent aucune analyse de valeurs.

Dans [23], Montagu et Jensen cherchent à inférer une forme de *frame condition* pour les langages fonctionnels purs, c'est-à-dire identifier des relations d'égalité entre des parties de valeurs algébriques. Une telle approche, si elle se confronte à des problématiques proches, ne vise pas à garder trace des valeurs des variables, non plus que des relations *numériques* entre celles-ci. Elle cherche plutôt à prouver des propriétés de sorte à en décharger les assistants de preuves et à effectuer des optimisations à la compilation. De plus, elle ne permet pas de manipuler des types algébriques récursifs.

Bautista et al. [2] proposent quant à eux une méthode d'abstraction relationnelle pour des valeurs algébriques contenant des données numériques, qu'ils enrichissent ensuite [3] en inférant des égalités structurelles entre champs non numériques. Cependant, cette approche, quoique plus précise que la nôtre dans des cas non récursifs, ne permet pas l'analyse des structures de données récursives tels les listes ou les arbres.

À notre connaissance, le seul analyseur statique de valeurs par interprétation abstraite pour un langage fonctionnel est [16], écrit en 2011. Il génère des contraintes avec sous-typage dont la vérification implique la sûreté du programme, puis les transforme en langage impératif de sorte à réutiliser des analyseurs existants. Cependant, dans cette approche, on perd très tôt la particularité fonctionnelle de notre langage source, ce qui risque d'impliquer une perte de précision. L'article mentionne une implémentation, mais celle-ci n'est pas disponible en ligne, aussi n'avons-nous pu comparer nos résultats.

6 Limitations et prolongements

Nos travaux permettent de mettre en évidence de nombreuses pistes vers une analyse statique de valeurs pour des langages fonctionnels d'ordre supérieur réalistes.

Pour l'heure, nous n'avons pas développé de méthode générique pour le polymorphisme. Pour cette raison, notre approche pour une fonction polymorphe diffère de celle d'une fonction monomorphe : on effectue l'analyse non plus lors de sa définition, mais lors de son application à des arguments de type connu. Ainsi, l'inférence du résumé de la fonction est réalisée dans une spécialisation du type de la fonction, et les méthodes pour les fonctions monomorphes s'appliquent. Cependant, travailler sur une analyse polymorphe permettrait d'améliorer les performances en ne réalisant qu'une seule analyse par variable, mais également d'obtenir des propriétés générales sur les variables polymorphes telles des relations d'égalité et d'inégalité, ainsi que des relations d'ordre dans un langage comme OCaml où tout type est muni d'une relation d'ordre total. Lever cette limitation serait un premier pas pour parvenir à l'analyse d'une bibliothèque.

De même, une méthode possible pour supporter l'ordre supérieur serait de procéder par injection (*inlining*) du corps des fonctions lorsque celles-ci sont appliquées. Ainsi, lorsqu'un argument d'une fonction est d'ordre supérieur, on diffère l'analyse de cette fonction jusqu'à ce qu'il puisse être spécialisé par une fonction connue, se ramenant ainsi

à une analyse du premier ordre. Cependant, notre analyse semble pouvoir s'adapter à l'ordre supérieur : en effet, en modélisant comme des relations, donc comme des points du domaine des polyèdres, des fonctions manipulant des entiers, fonctions comme valeurs sont réduites à un ensemble de points dans un espace vectoriel. La complexité et la précision de cette méthode serait à évaluer, tandis que l'étendre ou créer de nouveaux domaines pour les fonctions manipulant des objets algébriques sont des travaux futurs.

De plus, notre analyse se concentre pour l'heure sur un fragment pur de langage fonctionnel. Cela exclut de nombreuses fonctionnalités — tableaux, références, champs mutables — mais permet de simplifier l'analyse des relations entrées-sorties. Diverses pistes ont été envisagées pour étendre aux fonctionnalités impures. L'impureté d'une variable pouvant être identifiée par son type — `'a array`, `'a ref`, `mutable` — on peut choisir d'abstraire toute variable impure par la valeur maximale du domaine abstrait pour signifier qu'on ne connaît aucune information à son sujet, ou de les passer en paramètre de toute fonction susceptible de les manipuler. Pour améliorer la performance, on pourrait également identifier les fonctions dont les variables impures locales n'échappent pas à leur corps et ainsi utiliser les méthodes précédemment élaborées à moindre surcoût.

Un objectif à long terme serait enfin de s'appuyer sur cette analyse pour créer des domaines plus sophistiqués, capable d'établir automatiquement des propriétés plus fines sur les programmes. De telles utilisations de l'interprétation abstraite ont déjà été réalisées pour des langages impératifs, par Cousot et al. dans [7] pour des analyses de contenu de tableau, par Journault et Miné dans [17] pour prouver la correction de programme manipulant des matrices, ou encore avec la *shape analysis* [27], pour prouver des propriétés sophistiquées sur les listes et les arbres. Nous aimerions nous en inspirer pour créer des domaines capables d'exprimer des propriétés sur la taille des listes, la profondeur des structures de données, ou l'équilibre d'un arbre.

Conclusion

Ainsi, nous avons élaboré des méthodes d'analyse statique de valeur par interprétation abstraite dédiées aux langages fonctionnels. Les domaines abstraits sous-jacents sont relationnels, et permettent de traiter un fragment de langage fonctionnel du premier ordre supportant les types algébriques récursifs ainsi que les fonctions récursives les manipulant. Une partie de ces méthodes a été implémentées au sein de la plateforme MOPSA pour un fragment de OCaml et ont montré des résultats concluants sur un petit ensemble de programmes écrits à la main. Ce travail dégage également de nombreuses pistes vers l'obtention d'une analyse de valeurs précise et efficace pour un langage fonctionnel d'ordre supérieur.

Références

[1] Bagnara, R., Hill, P.M., Ricci, E. and Zaffanella, E., 2005. Precise widening operators for convex polyhedra. Science of Computer Programming, 58(1-2), pp.28-56.

[2] Bautista, S., Jensen, T. and Montagu, B., 2020, November. Numeric domains meet algebraic

data types. In Proceedings of the 9th ACM SIGPLAN International Workshop on Numerical and Symbolic Abstract Domains (pp. 12-16).

[3] Bautista, S., Jensen, T. and Montagu, B., 2022, December. Lifting Numeric Relational Domains to Algebraic Data Types. In Static Analysis : 29th International Symposium, SAS 2022, Auckland, New Zealand, December 5–7, 2022, Proceedings (pp. 104-134). Cham : Springer Nature Switzerland.

[4] Bertrane, J., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A. and Rival, X., 2015. Static analysis and verification of aerospace software by abstract interpretation. Foundations and Trends in Programming Languages, 2(2-3), pp.71-190.

[5] Bourdoncle, F., 1992. Abstract interpretation by dynamic partitioning. Journal of Functional Programming, 2(4), pp.407-435.

[6] Bühler, D., 2017. Structuring an abstract interpreter through value and state abstractions : eva, an evolved value analysis for frama-c (Doctoral dissertation, Université de Rennes 1).

[7] P. Cousot & N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In Proc. POPL'78, 84–96, ACM, 1978.

[8] Cousot, P. and Cousot, R., 1977. Static determination of dynamic properties of generalized type unions. ACM SIGOPS Operating Systems Review, 11(2), pp.77-94.

[9] Cousot, P. and Cousot, R., 1977, January. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (pp. 238-252).

[10] Cousot, P. and Cousot, R., 1994, May. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages). In Proceedings of 1994 IEEE International Conference on Computer Languages (ICCL'94) (pp. 95-112). IEEE.

[11] Cousot, P., Cousot, R. and Logozzo, F., 2011. A parametric segmentation functor for fully automatic and scalable array content analysis. ACM SIGPLAN Notices, 46(1), pp.105-118.

[12] Distefano, D., Fähndrich, M., Logozzo, F. and O'Hearn, P.W., 2019. Scaling static analyses at Facebook. Communications of the ACM, 62(8), pp.62-70.

[13] Filliâtre, J.C. and Paskevich, A., 2013, March. Why3—where programs meet provers. In European symposium on programming (pp. 125-128). Springer, Berlin, Heidelberg.

[14] Gopan, D., DiMaio, F., Dor, N., Reps, T., Sagiv, M., 2004. Numeric Domains with Summarized Dimensions, in : Jensen, K., Podelski, A. (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 512–529. https ://doi.org/10.1007/978-3-540-24730-2_38

[15] C. Gunter & D. Scott. Semantic domains. Vol. B of Handbook of Theoretical Computer Science, pp. 633–674. Elsevier, 1990.

[16] Jhala, R., Majumdar, R., & Rybalchenko, A. (2011, July). HMC : Verifying functional programs using abstract interpreters. In International Conference on Computer Aided Verification (pp. 470-485). Springer, Berlin, Heidelberg.

[17] Journault, M. and Miné, A., 2016, September. Static analysis by abstract interpretation of the functional correctness of matrix manipulating programs. In International Static Analysis Symposium (pp. 257-277). Springer, Berlin, Heidelberg.

[18] Journault, M., 2019. Analyse statique modulaire précise par interprétation abstraite pour la preuve automatique de correction de programmes et pour l'inférence de contrats (Doctoral

dissertation, Sorbonne université).

[19] Journault, M., Miné, A., Monat, R. and Ouadjaout, A., 2020. Combinations of reusable abstract domains for a multilingual static analyzer. In Working Conference on Verified Software : Theories, Tools, and Experiments (pp. 1-18). Springer, Cham.

[20] Krishnaswami, N., 2006. Separation logic for a higher-order typed language. In Workshop on Semantics, Program Analysis and Computing Environments for Memory Management, SPACE (Vol. 6, pp. 73-82).

[21] Liang, S. and Might, M., 2013, November. Entangled abstract domains for higher-order programs. In Proceedings of the 2013 Workshop on Scheme and Functional Programming, Washington, DC.

[22] Monat, R., 2021. Static type and value analysis by abstract interpretation of Python programs with native C libraries (Doctoral dissertation, Sorbonne Université).

[23] Montagu, B. and Jensen, T., 2020. Stable relations and abstract interpretation of higher-order programs. Proceedings of the ACM on Programming Languages, 4(ICFP), pp.1-30.

[24] Montagu, B. and Jensen, T., 2021, June. Trace-based control-flow analysis. In Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (pp. 482-496).

[25] Nielson, F. and Nielson, H.R., 1997, January. Infinitary control flow analysis : a collecting semantics for closure analysis. In Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (pp. 332-345).

[26] Rice, H. G., 1953, Classes of recursively enumerable sets and their decision problems. Transactions of the American Mathematical society, 74(2), 358-366.

[27] Sagiv, M., Reps, T. and Wilhelm, R., 2002. Parametric shape analysis via 3-valued logic. ACM Transactions on Programming Languages and Systems (TOPLAS), 24(3), pp.217-298.

[28] Smaragdakis, Y., Bravenboer, M. and Lhoták, O., 2011, January. Pick your contexts well : understanding object-sensitivity. In Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (pp. 17-30).

[29] Spoto, F., 2016, September. The Julia static analyzer for Java. In International Static Analysis Symposium (pp. 39-57). Springer, Berlin, Heidelberg.

[30] Swamy, N., Hriţcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.Y., Kohlweiss, M. and Zinzindohoue, J.K., 2016, January. Dependent types and multi-monadic effects in F. In Proceedings of the 43rd annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (pp. 256-270).

[31] Swamy, N., Weinberger, J., Schlesinger, C., Chen, J. and Livshits, B., 2013. Verifying higher-order programs with the Dijkstra monad. ACM SIGPLAN Notices, 48(6), pp.387-398.

[32] Vazou, N., Seidel, E.L., Jhala, R., Vytiniotis, D. and Peyton-Jones, S., 2014, August. Refinement types for Haskell. In Proceedings of the 19th ACM SIGPLAN international conference on Functional programming (pp. 269-282).

Annexe A Sémantique du langage fonctionnel

On définit en Fig 4 la sémantique concrète collectrice de notre langage fonctionnel.

Ainsi, la valeur d'une variable x dans un environnement σ est $\sigma(x)$, celle d'un entier n est ce même entier. L'expression fun $x_1 \dots x_n \to e$ est interprétée comme une fonction

$$\mathbb{E}[\![\cdot]\!] : \Sigma \to \mathcal{V}$$
$$\mathbb{E}[\![x \in \mathbb{V}]\!]\sigma = \sigma(x)$$
$$\mathbb{E}[\![n \in \mathbb{Z}]\!]\sigma = n$$
$$\mathbb{E}[\![\text{fun } x_1 \dots x_n \to e]\!]\sigma = \lambda a_1 \dots a_n.\mathbb{E}[\![e]\!]\sigma[x_1 \to a_1, \dots, x_n \to a_n]$$
$$\mathbb{E}[\![e_1 e_2 \dots e_{n+1}]\!]\sigma = \begin{cases} f[x_i \to \mathbb{E}[\![e_i]\!]\sigma] & \text{si } \mathbb{E}[\![e_1]\!]\sigma = \lambda x_1 \dots x_n.f \wedge \forall 1 \leq i \leq n, \mathbb{E}[\![e_i]\!]\sigma \in \mathcal{V}_0 \backslash \{\omega\} \\ \omega & \text{sinon} \end{cases}$$
$$\mathbb{E}[\![\text{if } e_1 \text{ then } e_2 \text{ else } e_3]\!]\sigma = \begin{cases} \mathbb{E}[\![e_2]\!]\sigma & \text{si } \mathbb{E}[\![e_1]\!]\sigma = z \in \mathbb{Z} \wedge z \neq 0 \\ \mathbb{E}[\![e_3]\!]\sigma & \text{si } \mathbb{E}[\![e_1]\!]\sigma = 0 \\ \omega & \text{sinon} \end{cases}$$
$$\mathbb{E}[\![\text{assert } e]\!]\sigma = \begin{cases} 1 & \text{si } \mathbb{E}[\![e]\!]\sigma = z \in \mathbb{Z} \wedge z \neq 0 \\ \omega & \text{sinon} \end{cases}$$
$$\mathbb{E}[\![\text{let } x = e_1 \text{ in } e_2]\!]\sigma = \begin{cases} \mathbb{E}[\![e_2]\!]\sigma[x \to \mathbb{E}[\![e_1]\!]\sigma] & \text{si } \mathbb{E}[\![e_1]\!]\sigma \neq \omega \\ \omega & \text{sinon} \end{cases}$$
$$\mathbb{E}[\![\text{let rec } x_f = e_1 \text{ in } e_2]\!]\sigma = \begin{cases} \mathbb{E}[\![e_2]\!]\sigma[x_f \to \text{lfp}(F_{x_f,x,g})] & \text{si } \mathbb{E}[\![e_1]\!]\sigma = \lambda x_1 \dots x_n.g \\ \omega & \text{sinon} \end{cases}$$
$$\text{avec } F_{x_f,x,g}(\phi) = \lambda y_1 \dots y_n.\mathbb{E}[\![g]\!]\sigma[x_i \to y_i, x_f \to \phi]$$
$$\mathbb{E}[\![e_1 \oplus e_2]\!]\sigma = \begin{cases} n_1 \oplus n_2 & \text{si } \mathbb{E}[\![e_1]\!]\sigma = n_1 \in \mathbb{Z} \text{ et } \mathbb{E}[\![e_2]\!]\sigma = n_2 \in \mathbb{Z} \\ \omega & \text{sinon} \end{cases}$$
$$\mathbb{E}[\![C(e_1, \dots, e_n)]\!]\sigma = \begin{cases} C(\mathbb{E}[\![e_1]\!]\sigma, \dots, \mathbb{E}[\![e_n]\!]\sigma) & \text{si } \forall 1 \leq i \leq n, \mathbb{E}[\![e_i]\!]\sigma \in \mathcal{V}_0 \backslash \{\omega\} \\ \omega & \text{sinon} \end{cases}$$
$$\mathbb{E}[\![\text{match } e_0 \text{ with } p_1 \to e_1 \mid \cdots \mid p_m \to e_m]\!]\sigma = \begin{cases} \mathbb{E}[\![e_1]\!]\sigma' & \text{si } b_{guard} = \textbf{true} \\ \mathbb{E}[\![e_1']\!]\sigma & \text{si } b_{guard} = \textbf{false} \wedge m > 1 \\ \omega & \text{sinon} \end{cases}$$
$$\text{avec } \begin{cases} \texttt{match}(\sigma, \mathbb{E}[\![e_0]\!]\sigma, p_1) = (\sigma', b_{guard}), & \text{cf. Fig. 5} \\ e_1' = \text{match } e_0 \text{ with } p_2 \to e_2 \mid \cdots \mid p_m \to e_m \end{cases}$$

Figure 4 – Sémantique fonctionnelle du langage.

mathématique, qui à des paramètres $a_1 \dots a_n$ renvoie l'objet mathématique correspondant à la sémantique de e dans un contexte où les variables x_i valent a_i. L'application s'interprète de la manière suivante : si e_1 s'interprète en une fonction, on évalue cette dernière avec pour arguments les interprétations des e_i si ceux-ci ne sont pas des fonctions. Sinon, ω est renvoyée. La sémantique du if est naturelle : si e_1 s'interprète en un entier non nul, la valeur est celle de l'interprétation de e_2, celle de e_3 s'il s'interprète en

$$\texttt{match}(\,\cdot\,,\,\cdot\,,\,\cdot\,) : \Sigma \times \mathcal{V} \times \Pi \to \Sigma \times \mathcal{B} \cup \{\omega\}$$
$$\texttt{match}(\sigma, \omega, p) = \sigma,\ \omega$$
$$\texttt{match}(\sigma, v, \_) = \sigma,\ \textbf{true}$$
$$\texttt{match}(\sigma, v, x \in \mathcal{V}) = \sigma[x \to v],\ \textbf{true}$$
$$\texttt{match}(\sigma, v, n_2 \in \mathbb{Z}) = \sigma, \begin{cases} n_1 = n_2 & \text{si } v = n_1 \in \mathbb{Z} \\ \textbf{false} & \text{sinon} \end{cases}$$
$$\texttt{match}(\sigma, C(v_1, \ldots, v_n), C(p_1, \ldots, p_n)) = \sigma_n,\ \bigwedge_{i \in [\![1,n]\!]} b_i$$
$$\text{avec } \begin{cases} \sigma_0 = \sigma \\ \forall i \in [\![1,n]\!], \sigma_{i+1}, b_{i+1} = \texttt{match}(\sigma_i, v_i, p_i) \end{cases}$$
$$\texttt{match}(\sigma, v, C(p_1, \ldots, p_n)) = \sigma,\ \textbf{false} \quad \text{avec } v \neq C(v_1, \ldots, v_n)$$
$$\texttt{match}(\sigma, v, p_1 \text{ when } e_1) = \sigma', b \wedge \mathbb{E}[\![e_1]\!]\sigma' \neq 0 \quad \text{avec } \texttt{match}(\sigma, v, p) = \sigma', b$$

Figure 5 – Définition de `match`.

0, une erreur sinon. L'assert s'interprète en 1 si e s'interprète en un entier non nul, une erreur sinon. Le let correspond à l'interprétation de e_2 dans un environnement où x a comme valeur l'interprétation de e_1 si celle-ci n'est pas une erreur, ω sinon.

Le let rec quant à lui associe à x_f l'interprétation suivante : si e_1 est une fonction, on considère la fonction $F_{x_f,x,g}$, qui à une fonction ϕ associe la fonction qui à y associe l'interprétation du corps g. Cette fonction, $F_{x_f,x,g}(\phi)$, est ainsi une meilleure approximation de x_f que ϕ. On associe alors à x_f le plus petit point fixe de $F_{x_f,x,g}$, défini sur les fonctions continues. Ce point fixe est bien défini [15].

Enfin, $e_1 \oplus e_2$ a pour valeur l'opération mathématique correspondant à $\oplus$ appliquée aux interprétations de e_1 et e_2 si toutes deux sont des entiers, ω sinon. $C(e_1, \ldots, e_n)$ s'interprète en interprétant chacune des expressions en paramètres du constructeur de type si celles-ci ne sont pas des fonctions et ne renvoient pas d'erreurs, ω sinon. La sémantique du match est plus complexe. On définit pour ce faire la fonction `match` en Fig. 5.

La fonction `match` prend en paramètre un environnement, une valeur et un motif. Elle renvoie un booléen, vrai si et seulement si la valeur peut correspondre au motif, ainsi que l'environnement dans lequel les variables du motif sont associées aux valeurs correspondantes de l'expression.

Ainsi, si le v est un joker _, la valeur peut correspondre au motif : le booléen est vrai et l'environnement est inchangé car aucune variable n'a été liée. Si le motif est une variable x, la valeur v peut également correspondre au motif dans un environnement où x est lié à v. Si le motif est un entier $n_2 \in \mathbb{Z}$, alors v correspond à n_2 si et seulement si n_1 et n_2 vus comme des entiers mathématiques sont égaux. Sinon, le booléen est faux.

Enfin, si le motif est sous la forme $C(p_1, \ldots, p_n)$ et la valeur de la forme $C(v_1, \ldots, v_n)$, la valeur correspond au motif si et seulement si tous les *bvi* peuvent correspondre aux p_i. L'environnement est alors l'environnement σ_n dans lequel si la conjonction des b_i est vrai, les v_i correspondent aux p_i. Comme $\forall i, j, fv(p_i) \neq fv(p_j)$, une variable n'a pas été liée plusieurs fois. Quant au motif p_1 when e_1, la valeur correspond au motif si et seulement si elle correspond à p_1 et e_1 s'interprète en une valeur non nulle.

Pour revenir à la sémantique du filtrage de motif, match e_0 with $p_1 \to e_1 \mid \ldots \mid p_m \to e_m$ est interprété de la manière suivante : si le booléen renvoyé par $\texttt{match}(\sigma, e_0, p_1)$ est vrai, c'est-à-dire si e_0 et p_1 peuvent correspondre, on évalue e_1 dans l'environnement σ' dans lequel e_0 et p_1 correspondent. Sinon, si $m > 1$, on ré-interprète le match en retirant le cas $p_1 \to e_1$. Enfin, si $m = 1$ et b_{guard} est faux, l'expression ne correspond à aucun motif et une erreur est renvoyée.

Intuitivement donc, l'expression est séquentiellement comparée aux différents motifs ; sitôt que celle-ci correspond à un motif p_i, l'expression e_i correspondante est évaluée dans le contexte résultant.

Annexe B Fonctions de transfert

On détaille ici les fonctions de transfert pour les constructions du langage.

$$\mathbb{E}^\sharp[\![x]\!](\epsilon, d) = \epsilon(x), (\epsilon, d)$$

$$\mathbb{E}^\sharp[\![n \in \mathbb{Z}]\!]\sigma^\sharp = n, \sigma^\sharp$$

$$\mathbb{E}^\sharp[\![\text{if } e_1 \text{ then } e_2 \text{ else } e_3]\!]\sigma^\sharp = \mathbb{E}^\sharp[\![e_2]\!](\mathcal{F}^\sharp[\![e_1^n]\!]\sigma_1^\sharp) \cup^\sharp \mathbb{E}^\sharp[\![e_3]\!](\mathcal{F}^\sharp[\![\neg e_1^n]\!]\sigma_1^\sharp)$$
$$\text{où } \mathbb{E}^\sharp[\![e_1]\!]\sigma^\sharp = e_1^n, \sigma_1^\sharp$$

$$\mathbb{E}^\sharp[\![\text{let } x = e_1 \text{ in } e_2]\!]\sigma^\sharp = \mathbb{E}^\sharp[\![e_2]\!]\sigma_1^\sharp[x \to e^\sharp] \quad \text{où } \mathbb{E}^\sharp[\![e_1]\!]\sigma^\sharp = e^\sharp, \sigma_1^\sharp$$

$$\mathbb{E}^\sharp[\![e_1 \oplus e_2]\!]\sigma^\sharp = x \oplus y, \sigma_2^\sharp[x \to e_1^n][y \to e_2^n], \quad \text{où } \mathbb{E}^\sharp[\![e_1]\!]\sigma^\sharp = e_1^n, \sigma_1^\sharp,\ \mathbb{E}^\sharp[\![e_2]\!]\sigma_1^\sharp = e_2^n, \sigma_2^\sharp$$

$$\mathbb{E}^\sharp[\![\text{assert } e]\!]\sigma^\sharp = \begin{cases} \{\omega\}, \sigma^\sharp & \text{si } \mathcal{F}^\sharp[\![e^n]\!]\sigma_0^\sharp \neq \bot,\ \mathbb{E}^\sharp[\![e]\!]\sigma^\sharp = e^n, \sigma_0^\sharp \\ \mathbb{E}^\sharp[\![1]\!]\sigma^\sharp & \text{sinon} \end{cases}$$

Figure 6 – Fonctions de transfert pour les constructions du langage.

Ainsi, l'interprétation abstraite d'une variable renvoie sa valeur dans l'environnement abstrait ainsi que l'environnement abstrait dans lequel ceci est vrai - en effet, si x est un entier, $\epsilon(x)$ est une variable numérique dont les valeurs possibles sont définies par d. L'interprétation de n renvoie l'entier n dans l'environnement inchangé. L'interprétation de if e_1 then e_2 else e_3 est l'union abstraite de l'interprétation de e_2 dans un environnement restreint dans lequel l'expression numérique associée à e_1 est nécessairement non nulle et de celles de e_3 dans un environnement restreint dans elle est nécessairement nulle. let $x = e_1$ in e_2 a comme interprétation abstraite celle de e_2 dans l'environnement dans lequel x est assigné à l'interprétation de $\mathbb{E}^\sharp[\![e_1]\!]\sigma^\sharp$ - par définition

de la notation $\sigma^\sharp[x \to y]$, l'opérateur d'assignation du domaine numérique peut être utilisé pour ce faire. $e_1 \oplus e_2$ a comme interprétation abstraite l'expression numérique $x \oplus y$, où x et y sont des variables fraîches, assignées dans l'environnement de retour aux interprétations de $\mathbb{E}^\sharp[\![e_1]\!]\sigma^\sharp$ et $\mathbb{E}^\sharp[\![e_2]\!]\sigma^\sharp$ - ce qui appelle l'assignation numérique. assert e s'interprète en $\Omega = \{\omega\}$ s'il existe un environnement restreint non vide dans lequel l'interprétation de e est non vide, 1 sinon.

Annexe C Définitions

C.1 Concrétisation

Définition C.1. Pour $e \in \mathcal{E}$, $\sigma^\sharp \in \Sigma^\sharp$, en notant $\mathbb{E}^\sharp[\![e]\!]\sigma^\sharp = (e^\sharp, (\epsilon, d))$, on a :

$$\gamma(\mathbb{E}^\sharp[\![e]\!]\sigma^\sharp) = \begin{cases} \{\ n \in \mathbb{Z} \mid \exists \delta \in (\mathbb{V} \to \mathbb{Z}), \delta(\epsilon(e)) = n \wedge \delta \in \gamma_{\mathbb{Z}}(d)\ \} & \text{si } e : \texttt{int} \\ \gamma_{\mathtt{t}}(e^\sharp) & \text{sinon} \end{cases}$$

$$\text{en notant } \epsilon(e) = \begin{cases} n & \text{si } e = n \in \mathbb{Z} \\ \epsilon(x) & \text{si } e = x \in \mathbb{V}_{\mathbb{Z}} \\ \epsilon(e_1) \oplus \epsilon(e_2) & \text{sinon} \end{cases}$$

Définition C.2. Pour $\sigma \in \Sigma$, $\sigma^\sharp = (\epsilon, d) \in \Sigma^\sharp$, on note $\sigma \in \sigma^\sharp \iff \forall x \in \mathbb{V} \backslash \mathbb{V}_{\mathbb{Z}}, \sigma(x) \in \gamma(\epsilon(x)) \wedge \sigma|_{\mathbb{Z}} \in \gamma_{\mathbb{Z}}(d)$

Lemme C.1. Si $\sigma \in \sigma^\sharp$, alors $\forall x, \sigma(x) \in \gamma(\sigma^\sharp(x), \sigma^\sharp)$.

Rappelons maintenant que pour $e \in \mathcal{E}, \sigma_p \in \mathcal{P}(\Sigma)$, on a $\mathcal{F}[\![e]\!]\sigma_p = \{\ \sigma \in \sigma_P \mid \mathbb{E}[\![e]\!]\sigma = n \in \mathbb{Z} \wedge n \neq 0\ \}$. La définition de la fonction de filtre $\mathcal{F}^\sharp$, la fonction de filtre dans l'abstrait, dépend du domaine choisi pour représenter les entiers et est fournie avec celui-ci. Elle vérifie la propriété :

Lemme C.2. Pour $e \in \mathcal{E}, \sigma^\sharp \in \Sigma^\sharp$, si $\sigma_p \subseteq \sigma^\sharp$, $\mathcal{F}[\![e]\!]\sigma_p \subseteq \gamma(\mathcal{F}^\sharp[\![e^n]\!]\sigma_0^\sharp)$, avec $\mathbb{E}^\sharp[\![e]\!]\sigma^\sharp = e^n, \sigma_0^\sharp$.

C.2 Union abstraite $\cup^\sharp$

Soient $(e_1^\sharp, (s_1, d_1)), (e_2^\sharp, (s_2, d_2)) \in \mathcal{V}^\sharp \times \Sigma^\sharp$. Alors :

$$(e_1^\sharp, (s_1, d_1)) \cup^\sharp (e_2^\sharp, (s_2, d_2)) = \begin{cases} (x, (s_1, d_1)[x \to e_1^\sharp] \cup^\sharp_{\Sigma^\sharp} (s_2, d_2)[x \to e_2^\sharp]) & \text{si } \begin{array}{l} e_1^n, e_2^n \in \mathcal{E}_{\mathbb{Z}}, \\ x \text{ fraîche} \end{array} \\ e_1^\sharp \cup^\sharp_{\mathcal{D}} e_2^\sharp, (s_1, d_i) \cup^\sharp_{\Sigma^\sharp} (s_2, d_2) & \text{sinon} \end{cases}$$

où $\cup^\sharp_{\Sigma^\sharp}$ est définie comme suit :

$$(s_1, d_1) \cup^\sharp_{\Sigma^\sharp} (s_2, d_2) = \begin{cases} x \to x_i, & \text{si } \begin{array}{l} x \in \mathbb{V}_\mathbb{Z}, x_i \text{ fraîche} \\ v_i = s_1(x) \cup^\sharp_\mathbb{Z} s_2(x) \end{array} \quad , d_0[x_i \to v_i] \\ x \to s_1(x) \cup_\mathcal{D} s_2(x) & \text{sinon} \end{cases}$$

avec $d_0 = d_1 \cup_{\mathcal{D}_\mathbb{Z}} d_2$

Lemme C.3. $\cup^\sharp$ est sûr.

C.3 Fonction $\texttt{match}^\sharp$

On définit ici $\texttt{match}^\sharp$, utilisé dans la définition de la sémantique abstraite.

$$\texttt{match}^\sharp(\,\cdot\,,\,\cdot\,,\,\cdot\,) : \Sigma^\sharp \times \mathcal{V}^\sharp \times \Pi \to \Sigma^\sharp \times \Sigma^\sharp$$
$$\texttt{match}^\sharp(\sigma^\sharp, v^\sharp, \_) = \sigma^\sharp, \bot$$
$$\texttt{match}^\sharp(\sigma^\sharp, v^\sharp, x \in \mathcal{V}) = \sigma^\sharp[x \to v^\sharp], \bot$$
$$\texttt{match}^\sharp(\sigma^\sharp, v^\sharp, n_2 \in \mathbb{Z}) = \mathcal{F}^\sharp[\![v^\sharp]\!]\sigma^\sharp, \mathcal{F}^\sharp[\![\neg v^\sharp]\!]\sigma^\sharp$$
$$\texttt{match}^\sharp(\sigma^\sharp, v^\sharp, C_i(p_1, \ldots, p_n)) = \sigma^\sharp_n, \sigma^\sharp_{\neg,n}$$
$$\text{avec } \begin{cases} v^\sharp = (g, \mathscr{C}) \in \mathcal{D}_\mathtt{t} \\ C_i \in \mathscr{C}, \sigma^\sharp_0 = \sigma^\sharp \\ \forall j \in [\![1, n]\!], \sigma^\sharp_{j+1}, \sigma^\sharp_{\neg,j+1} = \texttt{match}^\sharp(\sigma^\sharp_j, g_{i,j}, p_j) \end{cases}$$
$$\texttt{match}^\sharp(\sigma^\sharp, v^\sharp, p_1 \text{ when } e_1) = \mathcal{F}^\sharp[\![e_1^n]\!]\sigma^\sharp_1, \sigma^\sharp_{\neg,0}$$
$$\text{avec } \texttt{match}(\sigma^\sharp, v^\sharp, p) = \sigma^\sharp_0, \sigma^\sharp_{\neg,0}, \mathbb{E}^\sharp[\![e_1]\!]\sigma^\sharp_0 = e_1^n, \sigma^\sharp_1$$

Notons que la fonction de filtre $\mathcal{F}^\sharp$ est fournie par le domaine numérique. Étant donnée une expression numérique et un environnement abstrait, elle restreint cet environnement abstrait de sorte que l'expression numérique en paramètre soit non nulle.

Lemme C.4. Pour $\sigma, v, p \in \Sigma \times \mathcal{V} \times \Pi$, $\sigma^\sharp \in \Sigma^\sharp, v^\sharp \in \mathcal{V}^\sharp$ tel que $\sigma \in \sigma^\sharp, v \in \gamma(v^\sharp, \sigma^\sharp)$, alors en notant $\texttt{match}(\sigma, v, p) = \sigma_\mathtt{m}, b$ et $\texttt{match}^\sharp(\sigma^\sharp, v^\sharp, p) = \sigma^\sharp_\mathtt{m}, \sigma^\sharp_{\neg,\mathtt{m}}$, on a $b = \textbf{true} \implies \sigma_\mathtt{m} \in \sigma^\sharp_\mathtt{m}$ et $b = \textbf{false} \implies \sigma_\mathtt{m} \in \sigma^\sharp_{\neg,\mathtt{m}}$.

Démonstration. On procède par récurrence sur la syntaxe des motifs. On ne s'intéressera qu'au cas suivant :

- Si $p = C_l(p_1, \ldots, p_n)$.
  - Si $b = \textbf{true}$, alors par définition de `match`, $v = C_i(v_1, \ldots, v_n)$, et comme $v \in \gamma(v^\sharp, \sigma^\sharp) = \gamma_\mathtt{t}(v^\sharp, \sigma^\sharp)$, $v_i \in \gamma(g_{i,j}, \sigma^\sharp)$. Alors en notant $\sigma_0 = \sigma$ et pour $i \in [\![1, n]\!], \texttt{match}(\sigma_i, v_i, p_i) = \sigma_{i+1}, b_{i+1},$ et en constatant que $\forall i, b_i = \textbf{true}$, par une récurrence simple sur $i \in [\![1, n]\!]$, on prouve que $\sigma_i \in \sigma^\sharp_i$ (on applique successivement l'hypothèse de récurrence. Alors en particulier,
  - Si $b = \textbf{false}$, le raisonnement est le même en remplaçant $\sigma^\sharp_i$ par $\sigma^\sharp_{\neg,i}$.

Les autres cas sont plus simples, et utilisent la correction de $\mathcal{F}^\sharp$. □

Annexe D Preuves

D.1 Sûreté de $\cup_\tau^\sharp$ et $\cap_\tau^\sharp$

Démonstration. Étant donné un type algébrique $\mathtt{t}$, on veut prouver que l'opérateur de $\mathcal{D}_\mathtt{t}$ $\cup_\mathtt{n}^\sharp$ est une abstraction sûre de $\cup$, c'est-à-dire :

$$\forall (g_1, \mathscr{C}_1), (g_2\mathscr{C}_2) \in \mathcal{D}_\mathtt{t}, \gamma_\mathtt{t}((g_1, \mathscr{C}_1)) \cup \gamma_\mathtt{t}((g_2, \mathscr{C}_2)) \subseteq \gamma_\mathtt{t}((g_1, \mathscr{C}_1) \cup_\tau^\sharp (g_2, \mathscr{C}_2))$$

Rappelons que :

$$(g^1, \mathscr{C}_1) \ \cup_\mathtt{t}^\sharp \ (g^2, \mathscr{C}_2) = (g, \mathscr{C}_1 \cup \mathscr{C}_2) \quad \text{avec } g_{i,j} = g^1_{i,j} \ \cup_{\mathcal{D}_{i,j}} g^2_{i,j}$$

Soit maintenant $x \in \gamma_\mathtt{t}((\mathscr{C}_1, g_1)) \cup \gamma_\mathtt{t}((\mathscr{C}_2, g_2))$. Supposons sans perte de généralité que $x \in \gamma_\mathtt{t}((\mathscr{C}_1, g_1))$. Alors remarquons que $\mathscr{C}_1 \subseteq \mathscr{C}_1 \cup \mathscr{C}_2$ et que $\forall i, j g^1_{i,j} \leq_{i,j} g^2_{i,j}$. D'où $(\mathscr{C}_1, g_1) \sqsubseteq (\mathscr{C}_1 \cup \mathscr{C}_2, g)$. Donc par croissance de $\gamma_\mathtt{t}$, on a bien $\gamma_\mathtt{t}((g_1, \mathscr{C}_1)) \cup \gamma_\mathtt{t}((g_2, \mathscr{C}_2)) \subseteq \gamma_\mathtt{t}((g_1, \mathscr{C}_1) \cup_\tau^\sharp (g_2, \mathscr{C}_2))$.

La preuve de sûreté de l'opérateur $\cap_\tau^\sharp$ est similaire. □

D.2 Sûreté de l'élargissement pour les objets algébriques

Démonstration. Étant donné un type $\mathtt{t}$, on cherche à prouver que $\nabla_\mathtt{t}$ est bien un opérateur d'élargissement pour $\mathcal{D}_\mathtt{t}$. Rappelons que :

$$(\mathscr{C}_1, g_1) \nabla_\mathtt{t} (\mathscr{C}_2, g_2) = (g, \mathscr{C}_1 \cup \mathscr{C}_2) \quad \text{avec } g_{i,j} = g_1|_{i,j} \ \nabla_{\mathcal{D}_{i,j}} g_2|_{i,j}$$

- $\nabla_\mathtt{t}$ sur-approxime l'union. On note $X_{\leq h} = \{x \in X \mid h(x) \leq h\}$ avec $h(x)$ la profondeur de x. On prouve par récurrence sur h $\mathcal{P}(h)$: "$\forall (\mathscr{C}_1, g_1), (\mathscr{C}_2, g_2), \gamma_\mathtt{t}((\mathscr{C}_1, g_1))_{\leq h} \cup \gamma_\mathtt{t}((\mathscr{C}_2, g_2))_{\leq h} \subseteq \gamma_\mathtt{t}((\mathscr{C}_1, g_1) \nabla_\mathtt{t} \gamma_\mathtt{t}((\mathscr{C}_2, g_2))_{\leq h}$". Pour $h = 0$, les ensembles sont vides ou contiennent uniquement des feuilles (aucun champ récursif) et le raisonnement du cas $\mathtt{ti,j} \neq t$ ci-dessous s'applique.
  Soit $h \in \mathbb{N}$, $(g_1, \mathscr{C}_1), (g_2, \mathscr{C}_2) \in \mathcal{D}_\mathtt{t}$. Soit $x \in \gamma_\mathtt{t}(((g_1, \mathscr{C}_1))_{\leq h+1} \cup \gamma_\mathtt{t}((g_2, \mathscr{C}_2))_{\leq h+1}$. Supposons sans perte de généralités que $x \in \gamma_\mathtt{t}(((g_1, \mathscr{C}_1))_{\leq h+1}$.
  Alors $x = C_i(x_{i,1}, \ldots, x_{i,n_i})$, avec $C_i \in \mathscr{C}_1$, et $h(x) \leq h + 1$.
  - Supposons $\mathtt{ti,j} \neq \mathtt{t}$. On a $x_{i,j} \in \gamma_{\tau_{i,j}}(g_{1,i,j})$. Comme $\nabla_{\mathcal{D}_{i,j}}$ est un opérateur d'élargissement, on a $\gamma_{\tau_{i,j}}(g_{1,i,j}) \cup^\sharp_{i,j} \gamma_{\tau_{i,j}}(g_{2,i,j}) \subseteq \gamma_{\tau_{i,j}}(g_{1,i,j} \nabla_{i,j} g_{2,i,j})$, donc $\gamma_{\tau_{i,j}}(g_{1,i,j}) \subseteq \gamma_{\tau_{i,j}}(g_{1,i,j} \nabla_{i,j} g_{2,i,j})$. On en déduit :
    $x_{i,j} \in \gamma_{\tau_{i,j}}(g_1[i,j] \ \nabla_{\mathcal{D}_{i,j}} g_2[i,j]) = \gamma_{\tau_{i,j}}(g_{i,j})$.
  - Supposons $\mathtt{ti,j} = \mathtt{t}$. On a alors $x_{i,j} \in \gamma_\mathtt{t}(g_1, g_{1,i,j}))_{\leq h}$. Par hypothèse de récurrence, on a $\gamma_\mathtt{t}((\mathscr{C}_1, g_1))_{\leq h} \cup \gamma_\mathtt{t}((\mathscr{C}_2, g_2))_{\leq h} \subseteq \gamma_\mathtt{t}((\mathscr{C}_1, g_1) \nabla_\mathtt{t} \gamma_\mathtt{t}((\mathscr{C}_2, g_2))_{\leq h}$, d'où $\gamma_\mathtt{t}((\mathscr{C}_1, g_1))_{\leq h} \subseteq \gamma_\mathtt{t}((\mathscr{C}_1, g_1) \nabla_\mathtt{t} \gamma_\mathtt{t}((\mathscr{C}_2, g_2))_{\leq h}$, donc $x \in \gamma_\mathtt{t}((g, \mathscr{C}_1 \cup \mathscr{C}_2))_{\leq h}$.

  Ainsi, $\mathcal{P}(h+1)$ est vrai. Cela prouve donc la propriété pour tout $h \in \mathbb{N}$. Alors on a bien $\nabla_\mathtt{t}$ sur-approxime l'union : $\gamma_\mathtt{t}((\mathscr{C}_1, g_1)) \cup \gamma_\mathtt{t}((\mathscr{C}_2, g_2)) \subseteq \gamma_\mathtt{t}((\mathscr{C}_1, g_1) \nabla_\mathtt{t} \gamma_\mathtt{t}((\mathscr{C}_2, g_2))$

- $\nabla_{\mathtt{t}}$ assure la convergence en temps fini. Considérons la suite $((g_n, \mathscr{C}_n)))_{n\in\mathbb{N}}$. On cherche à prouver que la suite

$$((g_n^\nabla, C_n^\nabla))_{n\in\mathbb{N}} : \begin{cases} (g_0^\nabla, C_0^\nabla) = (g_0, \mathscr{C}_0) \\ (g_{n+1}^\nabla, C_{n+1}^\nabla) = (g_n^\nabla, C_n^\nabla)\nabla_{\mathtt{t}}(g_{n+1}, \mathscr{C}_{n+1}) \end{cases}$$

est stationnaire.
Pour i, j, on remarque que $g_n^\nabla|_{i,j} = \underset{1\le l\le n}{\nabla_{i,j}} g_l|_{i,j}$. Or $\nabla_{i,j}$ est un opérateur d'élargissement, donc cette suite est stationnaire. Soit $n_{i,j}$ l'indice à partir duquel cette suite est stationnaire. Notons $n_g = \max_{i,j}\ n_{i,j}$. Alors à partir du rang n_g, toutes les suites $(g_n^\nabla|_{i,j})_{n\in\mathbb{N}}$ sont stationnaires. Cela implique que la suite $(g_n^\nabla)_{n\in\mathbb{N}}$ est stationnaire à partir du rang n_g. Enfin, $\cup$ est un opérateur d'élargissement dans $\mathcal{P}(\mathbb{C}_{\mathtt{t}})$ donc il existe un rang n_C à partir duquel $(\mathscr{C}_n^\nabla)_{n\in\mathbb{N}}$ est stationnaire. Ainsi, à partir du rang $N = \max(n_h, n_C)$, $((g_n^\nabla, C_n^\nabla))_{n\in\mathbb{N}}$ est stationnaire.

Donc $\nabla_{\mathtt{t}}$ est bien un widening.

□

D.3 Sûreté de la sémantique abstraite

Démonstration. On cherche à prouver la sûreté de la sémantique abstraite $\mathbb{S}^\sharp$, c'est-à-dire

$$\forall\sigma\in\Sigma, \forall e\in\mathcal{E}, \sigma\in\sigma^\sharp \implies \mathbb{E}[\![e]\!]\sigma \in \gamma(\mathbb{E}^\sharp[\![e]\!]\sigma^\sharp) \wedge \sigma\in\sigma_0^\sharp \quad \text{avec } \mathbb{E}^\sharp[\![e]\!]\sigma^\sharp = e^\sharp, \sigma_0^\sharp$$

On effectue la preuve par récurrence sur la syntaxe de $e \in \mathcal{E}$.

- Soit e une expression telle qu'une sous-expression e_0 renvoie une erreur. Par hypothèse sur la sémantique abstraite, on a aussi $\mathbb{E}^\sharp[\![e]\!]\sigma^\sharp = \Omega$. On suppose l'expression bien typée, donc les seules erreurs proviennent des échecs de filtrage et des échecs d'assertion. Par hypothèse de récurrence, $\mathbb{E}[\![e_0]\!]\sigma \in \gamma(\mathbb{E}^\sharp[\![e_0]\!]\sigma^\sharp) = \gamma(\Omega) = \{\omega\}$, l'interprétation de e_0 renvoie une erreur, donc l'interprétation concrète de e renvoie une erreur. On a donc bien $\mathbb{E}[\![e]\!]\sigma \in \gamma(\mathbb{E}^\sharp[\![e]\!]\sigma^\sharp)$, et $\sigma \in \sigma_0^\sharp = \sigma^\sharp$. Par la suite, on supposera donc qu'aucune sous-expression ne renvoie d'erreur.

- Si $e = x$, alors $\mathbb{E}[\![e]\!]\sigma = \sigma(x)$, et comme $\sigma \in \sigma^\sharp$, on a $\sigma(x) \in \gamma(\sigma^\sharp(x), \sigma^\sharp) = \gamma(\mathbb{E}^\sharp[\![x]\!]\sigma^\sharp)$ et $\sigma \in \sigma_0^\sharp = \sigma^\sharp$.

- Si $e = z \in \mathbb{Z}$, alors $\mathbb{E}[\![e]\!]\sigma = z$. On a $\mathbb{E}^\sharp[\![z \in \mathbb{Z}]\!](\epsilon, d) = z, (\epsilon, d)$. Or, $\gamma(n, (\epsilon, d)) = \{\ n \in \mathbb{Z} \mid \exists\delta \in \mathcal{P}(\mathbb{V} \to \mathbb{Z}), \delta(z) = n \wedge \delta \in \gamma_{\mathbb{Z}}(d_0)\ \} = \{z\}$. D'où $\mathbb{E}[\![z \in \mathbb{Z}]\!]\sigma \in \gamma(\mathbb{E}^\sharp[\![z \in \mathbb{Z}]\!]\sigma^\sharp)$ et $\sigma \in \sigma_0^\sharp = \sigma^\sharp$.

- Si $e = e_1 \oplus e_2$, on a $\mathbb{E}^\sharp[\![e_1 \oplus e_2]\!]\sigma^\sharp = x \oplus y, \sigma_2^\sharp[x \to e_1^n][y \to e_2^n] = e^\sharp, \sigma^\sharp$, avec $\begin{cases}\mathbb{E}^\sharp[\![e_1]\!]\sigma^\sharp = e_1^n, \sigma_1^\sharp \\ \mathbb{E}^\sharp[\![e_2]\!]\sigma_1^\sharp = e_2^n, (\epsilon_2, d_2)\end{cases}$
  Alors $\gamma(\mathbb{E}^\sharp[\![e_1 \oplus e_2]\!]\sigma^\sharp) = \{n \in \mathbb{Z} \mid \exists \delta \in \mathcal{P}(\mathbb{X}^n), \delta(v_x) \oplus \delta(v_y) = n \wedge \delta \in \gamma_{\mathbb{Z}}(d)\}$.
  Par hypothèse de récurrence, $\begin{cases}\mathbb{E}[\![e_1]\!]\sigma \in \gamma(\mathbb{E}^\sharp[\![e_1]\!]\sigma^\sharp) \wedge \sigma \in \sigma_1^\sharp \\ \mathbb{E}[\![e_2]\!]\sigma \in \gamma(\mathbb{E}^\sharp[\![e_2]\!]\sigma^\sharp) \wedge \sigma \in \sigma_2^\sharp\end{cases}$.
  On en déduit que $\sigma|_{\mathbb{Z}} \in \gamma_{\mathbb{Z}}(d_2)$. Enfin, par correction de l'assignation, on a $\sigma' = \sigma|_{\mathbb{Z}}[v_x \to \sigma(e_1^n)][v_y \to \sigma(e_2^n)] \in \sigma_2^\sharp[x \to e_1^n][y \to e_2^n] = (\epsilon, d)$. Donc $\sigma' \in \gamma_{\mathbb{Z}}(d)$, $\sigma' \in \mathbb{V} \to \mathbb{Z}$ et $\sigma'(v_x \oplus v_y) = \sigma'(v_x) \oplus \sigma'(v_y) = \sigma'(x) + \sigma'(y) = \mathbb{E}[\![e_1]\!]\sigma \oplus \mathbb{E}[\![e_2]\!]\sigma$. D'où $\mathbb{E}[\![e_1 \oplus e_2]\!]\sigma = \mathbb{E}[\![e_1]\!]\sigma \oplus \mathbb{E}[\![e_2]\!]\sigma \in \mathbb{E}^\sharp[\![e_1 \oplus e_2]\!]\sigma^\sharp$. Enfin, comme $\forall i, \sigma \in \sigma_i^\sharp$, $\sigma \in \sigma_0^\sharp \subseteq \sigma_0^\sharp[r.i.j \to h_{i,j}$. Enfin, $\sigma \leq \sigma' \in \sigma_0^\sharp$.
- Si $e = C_l(e_1, \ldots, e_n)$, on a $\mathbb{E}[\![e]\!]\sigma = C_l(\mathbb{E}[\![e_1]\!]\sigma, \ldots, \mathbb{E}[\![e_n]\!]\sigma)$ (puisque l'expression est bien typée par hypothèse). Rappelons que :

$$\mathbb{E}^\sharp[\![C_l(e_1, \ldots, e_n)]\!]\sigma^\sharp = (g, \{C_l\}), \sigma_0^\sharp[r.i.j \to h_{i,j}]$$
$$\text{avec } \begin{cases}\mathbb{E}^\sharp[\![e_i]\!]\sigma^\sharp = v_i, \sigma_i^\sharp,\ (s, \mathscr{C}) = \bigcup\limits_{e_i:\mathtt{t}}^\sharp v_i,\ \sigma_0^\sharp = \bigsqcup\limits_i^\sharp \sigma_i^\sharp \\ h_{i,j} = \begin{cases} s_{l,j} \cup_{\mathcal{D}_{l,j}} v_j & \text{si } i = l \wedge \neg(\tau_{l,j} : \mathtt{t}) \\ s_{i,j} & \text{sinon}\end{cases} \\ g_{i,j} = \begin{cases} r.i.j & \text{si } \tau_{i,j} : \mathtt{int} \\ h_{i,j} & \text{sinon}\end{cases}\end{cases}$$

  On a $C_l \in \{C_l\}$. Remarquons que si $\mathtt{ti}, \mathtt{j} = \mathtt{int}$, on a $\gamma(g_{i,j}, \sigma^\sharp) = \gamma(r.i.j, \sigma^\sharp) = \gamma(h_{i,j}, \sigma^\sharp)$ par correction de l'assignation, on peut sans perte de généralités considérer $g_{i,j} = h_{i,j}$ dans la suite. Procédons maintenant par disjonction de cas.
  — Soit $j \in [\![1, n_l]\!]$ tels que $\mathtt{tl}, \mathtt{j} \neq \mathtt{t}$. Par hypothèse de récurrence, on sait que $\mathbb{E}[\![e_j]\!]\sigma \in \gamma(\mathbb{E}^\sharp[\![e_j]\!]\sigma^\sharp)$. Or, on sait que $g_{l,j} = s_{l,j} \cup_{\mathcal{D}_{i,j}} \mathbb{E}[\![e_j]\!]\sigma$, d'où $g_{i,j} \subseteq \mathbb{E}^\sharp[\![e_j]\!]\sigma^\sharp$, d'où par croissance de $\gamma_\mathtt{t}$, $\gamma_\mathtt{t}(g_{i,j}) \subseteq \gamma_\mathtt{t}(\mathbb{E}^\sharp[\![e_j]\!]\sigma^\sharp)$. On obtient donc $\mathbb{E}[\![e_j]\!]\sigma \in \gamma_{l,j}(g_{l,j})$.
  — Soit $i \in [\![1, n_l]\!]$ tels que `ti,j = t`. On a toujours $\mathbb{E}[\![e_j]\!]\sigma \in \gamma(\mathbb{E}^\sharp[\![e_j]\!]\sigma^\sharp)$. Notons $(\mathscr{C}_j, s_j) = \mathbb{E}^\sharp[\![e_j]\!]\sigma^\sharp$. Remarquons alors que $s_j \sqsubseteq s \sqsubseteq g$ On a alors $(\mathscr{C}_j, s_j) \sqsubseteq (g_{i,j}, g)$, donc par croissance de $\gamma_\mathtt{t}$, $\gamma_\mathtt{t}((\mathscr{C}_j, s_j)) \subseteq \gamma_\mathtt{t}((g_{i,j}, g))$, c'est-à-dire $\gamma(\mathbb{E}^\sharp[\![e_j]\!]\sigma^\sharp) \subseteq \gamma_\mathtt{t}((g_{i,j}, g))$, d'où $\mathbb{E}[\![e_j]\!]\sigma \in \gamma_\mathtt{t}((g_{i,j}, g))$.
  On a donc bien $C_l(\mathbb{E}[\![e_1]\!]\sigma, \ldots, \mathbb{E}[\![e_n]\!]\sigma) \in \gamma(\mathbb{E}^\sharp[\![e]\!]\sigma^\sharp)$. De plus, $\forall i, \sigma \in \sigma_i^\sharp \subseteq \sigma_0^\sharp \subseteq \sigma_0^\sharp[r.i.j \to h_{i,j}]$

- Si $e =$ if e_1 then e_2 else e_3, on a :

$$\mathbb{E}[\![\text{if } e_1 \text{ then } e_2 \text{ else } e_3]\!]\sigma = \begin{cases}\mathbb{E}[\![e_2]\!]\sigma & \text{si } \mathbb{E}[\![e_1]\!]\sigma = 0 \\ \mathbb{E}[\![e_3]\!]\sigma & \text{si } \mathbb{E}[\![e_1]\!]\sigma = z \in \mathbb{Z} \wedge z \neq 0\end{cases}$$

Par disjonction de cas :
— Supposons $\mathbb{E}[\![e_1]\!]\sigma = n \in \mathbb{Z}$ et $n \neq 0$, notons $\mathbb{E}^\sharp[\![e_1]\!]\sigma^\sharp = e^n, \sigma_1^\sharp$. Alors par le lemme C.2, $\sigma \in \mathcal{F}[\![e]\!]\{\sigma\} \subseteq \gamma(\mathcal{F}^\sharp[\![e^n]\!]\sigma_1^\sharp)$. Donc par hypothèse de récurrence, en notant $\mathbb{E}^\sharp[\![e_2]\!](\mathcal{F}^\sharp[\![e^n]\!]\sigma_1^\sharp) = e_2^\sharp, \sigma_2^\sharp$ et $\mathbb{E}[\![e_2]\!]\sigma \in \gamma(e_2^\sharp, \sigma_2^\sharp)$, d'où par croissance de γ, $\mathbb{E}[\![e_2]\!]\sigma \in \gamma(\mathbb{E}^\sharp[\![\text{if } e_1 \text{ then } e_2 \text{ else } e_3]\!]\sigma^\sharp)$, et par définition de $\cup^\sharp$, $\sigma \in \sigma_2^\sharp \subseteq \sigma_0^\sharp$
— Si $\mathbb{E}[\![e_i]\!] = 0$, en remplaçant e_1 par $\neg e_1$, le raisonnement est identique.

- Le cas du assert est identique à celui du if.

- Si $e = \text{match } e_0 \text{ with } p_1 \to e_1 \mid \cdots \mid p_m \to e_m$, alors notons $\texttt{match}(\sigma, \mathbb{E}[\![e_0]\!]\sigma, p_1) = (\sigma', b_{guard})$ et
$\texttt{match}^\sharp(\sigma^\sharp, \mathbb{E}^\sharp[\![e_0]\!]\sigma^\sharp, p) = \sigma_{\texttt{m}}^\sharp, \sigma_{\neg,\texttt{m}}^\sharp$. Procédons par disjonction de cas :
— Si $b_{guard} = \textbf{true}$, alors $\mathbb{E}[\![e]\!]\sigma = \mathbb{E}[\![e_1]\!]\sigma'$. Or, d'après le lemme C.4, $\sigma' \in \sigma_{\texttt{m}}^\sharp$. Donc $\mathbb{E}[\![e_1]\!]\sigma' \in \mathbb{E}^\sharp[\![e_1]\!]\sigma_{\texttt{m}}^\sharp$ par hypothèse de récurrence, et comme $\sigma_{\texttt{m}}^\sharp \neq \emptyset$ (il contient σ), on a $\mathbb{E}^\sharp[\![e_1]\!]\sigma_{\texttt{m}}^\sharp \subseteq \mathbb{E}^\sharp[\![e_1]\!]\sigma_{\texttt{m}}^\sharp \cup^\sharp \mathbb{E}^\sharp[\![e_1']\!]\sigma_{\neg,\texttt{m}}^\sharp = \mathbb{E}^\sharp[\![e]\!]\sigma^\sharp$, on a bien $\mathbb{E}[\![e]\!]\sigma \in \gamma(\mathbb{E}^\sharp[\![e]\!]\sigma^\sharp)$. Toujours par définition de $\cup^\sharp$, on a bien $\sigma \in \sigma_0^\sharp$.
— Sinon, si $b_{guard} = \textbf{false}$ et $m > 1$, alors par le lemme C.4, $\sigma' \in \sigma_{\neg,\texttt{m}}^\sharp$. Alors par hypothèse de récurrence, en notant $e_1' = \text{match } e_0 \text{ with } p_2 \to e_2 \mid \cdots \mid p_m \to e_m$, on a $\mathbb{E}[\![e_1']\!]\sigma \in \gamma(\mathbb{E}^\sharp[\![e_1']\!]\sigma^\sharp)$ et comme $m > 1$, on a bien : $\mathbb{E}^\sharp[\![e_1']\!]\sigma^\sharp \subseteq \mathbb{E}^\sharp[\![e_1]\!]\sigma_{\texttt{m}}^\sharp \cup^\sharp \mathbb{E}^\sharp[\![e_1']\!]\sigma_{\neg,\texttt{m}}^\sharp = \mathbb{E}^\sharp[\![e]\!]\sigma^\sharp$. De même, $\sigma \in \sigma_0^\sharp$.
— Enfin, si $b_{guard} = \textbf{false}$ et $m = 1$, alors $\sigma' \in \sigma_{\neg,1}^\sharp$, donc $\sigma_{\neg,1}^\sharp \neq \emptyset$, donc on appelle récursivement lanalyce avec $m = 0$ d'où $\mathbb{E}^\sharp[\![e]\!]\sigma^\sharp \subseteq \Omega$ donc on a bien $\omega \in \Omega$.

- Si $e = \text{let } x = e_1 \text{ in } e_2$, alors en notant $\mathbb{E}^\sharp[\![e_1]\!]\sigma^\sharp = e^\sharp, \sigma^\sharp$, puisque $\mathbb{E}[\![e_1]\!]\sigma \in \gamma(\mathbb{E}^\sharp[\![e_1]\!]\sigma^\sharp)$ et $\sigma \in \sigma^\sharp$, on a $\sigma[x \to \mathbb{E}[\![e_1]\!]\sigma] \in \sigma^\sharp[x \to e^\sharp]$ par correction de l'assignation, donc par hypothèse de récurrence, $\mathbb{E}[\![e_2]\!]\sigma[x \to \mathbb{E}[\![e_1]\!]\sigma] \in \gamma(\mathbb{E}^\sharp[\![e_2]\!]\sigma^\sharp[x \to e^\sharp])$ et $\sigma[x \to \mathbb{E}[\![e_1]\!]\sigma] \subseteq \sigma^\sharp[x \to e^\sharp]$, d'où $\mathbb{E}[\![e]\!]\sigma \in \mathbb{E}^\sharp[\![e]\!]\sigma^\sharp$ et $\sigma \in \sigma_0^\sharp$.

- Si $e = e_0 \dots e_n$. Comme e est bien typé, $\mathbb{E}^\sharp[\![e_0]\!]\sigma^\sharp = (x_1, \dots, x_n, e^\sharp, \sigma_0^\sharp)$. Notons que par hypothèse de récurrence et par construction de l'abstraction de fun, $\sigma \in \sigma_0^\sharp$. Notons $\mathbb{E}^\sharp[\![e_i]\!]\sigma^\sharp = e_i^\sharp, \sigma_i^\sharp$ Par hypothèse de récurrence, on a également $\forall i, \mathbb{E}[\![e_i]\!]\sigma^\sharp \in \gamma(e_i^\sharp, \sigma_i^\sharp)$. Or, par hypothèse de récurrence, $f = \mathbb{E}[\![e_0]\!]\sigma \in \gamma(\mathbb{E}^\sharp[\![e_0]\!]\sigma^\sharp)$, par définition de la concrétisation pour les abstractions de fonction, on en conclut que $\mathbb{E}[\![e]\!] = f(\mathbb{E}[\![e_1]\!]\sigma, \dots, \mathbb{E}[\![e_n]\!]\sigma) \in \gamma(e^\sharp, \sigma_0^\sharp[x_i = e_i^\sharp]) = \gamma(\mathbb{E}^\sharp[\![e]\!]\sigma^\sharp)$. Comme $x_i \notin \sigma$, on a également $x_i \in \sigma_0^\sharp[x_i = e_i^\sharp]$.

- Si $e = \text{fun } x_1 \dots x_n \to e_0$. On a $\mathbb{E}^\sharp[\![e]\!]\sigma^\sharp = (x_1, \dots, x_n, \mathbb{E}^\sharp[\![e_0]\!]\sigma^\sharp[x_1 \to \top, \dots, x_n \to$

$\top]), \sigma^\sharp)$ et $\mathbb{E}[\![e]\!]\sigma = \lambda a_1 \dots a_n.\mathbb{E}[\![e_0]\!]\sigma[x_i \to a_i]$. Soit $a_1 : \tau_1, \dots, a_n : \tau_n$ et $v_1 \in \mathcal{D}_1, \dots, v_n \in \mathcal{D}_n$ tels que $a_i \in \gamma(v_i, \sigma_i^\sharp)$. On a $f(a_1, \dots, a_n) = \mathbb{E}[\![e_0]\!]\sigma[x_i \to a_i]$. On note $\mathbb{E}^\sharp[\![e_0]\!]\sigma^\sharp[x_i \to \top]) = e^\sharp, \sigma_1^\sharp$. Remarquons que $\sigma[x_i \to a_i] \in \sigma^\sharp[x_i \to \top]$. Alors par hypothèse de récurrence $\mathbb{E}[\![e_0]\!]\sigma[x_i \to a_i] \in \gamma(\mathbb{E}^\sharp[\![e_0]\!]\sigma^\sharp[x_i \to \top]) = \gamma(e^\sharp, \sigma_1^\sharp)$ et $\sigma[x_i \to a_i] \in \sigma_1^\sharp$, que $a_i \in \gamma(v_i, \sigma_i^\sharp)$, on a $\sigma[x_i \to a_i] \in \sigma_1^\sharp[x_i = v_i]$, donc on conclut que $\mathbb{E}[\![e_0]\!]\sigma[x_i \to a_i] \in \gamma(e^\sharp, \sigma_1^\sharp[x_i = v_i])$. On obtient donc bien $f \in \gamma(\mathbb{E}^\sharp[\![e]\!]\sigma^\sharp)$. On a $\sigma \in \sigma_0^\sharp = \sigma^\sharp$.

- Si $e = \text{let rec } x_f = e_1 \text{ in } e_2$. On a :

$$\mathbb{E}[\![\text{let rec } x_f = e_1 \text{ in } e_2]\!]\sigma = \begin{cases} \mathbb{E}[\![e_2]\!]\sigma[x_f \to \text{lfp}(F_{x_f,x,g})] & \text{si } \mathbb{E}[\![e_1]\!]\sigma = \lambda x_1 \dots x_n.g \\ \omega & \text{sinon} \end{cases}$$

avec $F_{x_f,x,g}(\phi) = \lambda y.\mathbb{E}[\![g]\!]\sigma[x \to y, x_f \to \phi]$

On sait que $\text{lfp}(F_{x_f,x,g})$ vaut $\bigcup_{n\in\mathbb{N}}(F^n_{x_f,x,g}(\bot))$. Remarquons qu'on peut écrire $\lambda y.\mathbb{E}[\![g]\!]\sigma[x \to y, x_f \to \phi] = \mathbb{E}[\![e_1]\!]\sigma[x_f \to \phi]$. Alors étant donné v, par hypothèse de récurrence, pour $v \in \gamma(v^\sharp)$, on a $F_{x_f,x,g}(v) = \mathbb{E}[\![e_1]\!]\sigma[f \to v] \in \gamma(\mathbb{E}^\sharp[\![e_1]\!]\sigma^\sharp[x_f \to v^\sharp]) = \gamma(F^\sharp(v^\sharp))$.
En conséquence, en remarquant que $(x_1, \dots, x_n, \bot, \sigma^\sharp[x_1, \dots, x_n]) = \bot$ dans le domaine des fonctions, on a bien $\bigcup_{n\in\mathbb{N}}(F^n_{x_f,x,g}(\bot)) \in \gamma(\bigcup^\sharp_{n\in\mathbb{N}}((F^\sharp)^n(\bot)))$ par sûreté de $\cup^\sharp$. Comme ∇ est un opérateur d'élargissement, donc cette opération converge et sur-approxime $\bigcup^\sharp$, on a donc $\text{lfp}(F_{x_f,x,g}) \in \gamma(\nabla^\sharp_{n\in\mathbb{N}}.((F^\sharp)^n(\bot)))$.
D'où $\sigma[x_f \to \text{lfp}(F_{x_f,x,g})] \in \sigma^\sharp[x_f \to \nabla^\sharp_{n\in\mathbb{N}}.((F^\sharp)^n(\bot))]$. Le raisonnement qui suit est similaire à celui du let.

□

Annexe E Correspondance de Galois

Définition E.1. Si $\forall i, j,\ \mathcal{T}_{\texttt{ti,j}} \underset{\alpha_{i,j}}{\overset{\gamma_{\tau_{i,j}}}{\leftrightarrows}} \mathcal{D}_{i,j}$, on peut définir α récursivement de la manière suivante :

$$\alpha(\texttt{Ci}(x_{i,1}, \dots, x_{i,n_i}) = \left(\underset{\substack{1\le i'\le n \\ 1\le j\le n_i}}{\times} \begin{cases} \alpha_{i,j}(x_{i,j}) & \text{si } i' = i \wedge \texttt{ti,j} \neq \texttt{t} \\ \bot & \text{sinon} \end{cases} \times \{\texttt{Ci}\} \right) \underset{\texttt{ti,j}\neq\texttt{t}}{\cup^\sharp} \alpha(x_{i,j})$$

On a alors la correspondance de Galois $\mathcal{T}_\texttt{t} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{D}_\texttt{t}$.

Annexe F Listings des benchmarks

```
let x = Cons(1, Cons(2, Cons(3, Nil))) in
let y = Nil in
let z = Cons(4, x) in
```

```
x : { Cons }          x_1,1 : [1, 3]          x_1,2 : { Nil, Cons }
y : { Nil }
z : { Cons }          y_1,1 : [1, 4]          z_1,2 : { Nil, Cons }
```

Listing 1: `list.ml`

```
type tree = Node of tree * int * tree | Leaf of int
let x = Node(Node(Leaf(250), 100, Leaf(251)), 1, Leaf(252)) in
```

```
x: { Node }   x_1,1: { Node, Leaf }   x_1,2: [1, 100]   x_1,3 {Leaf}
              x_2,1: [250,252]
```

Listing 2: `tree.ml`

```
let x = match Cons(1, Nil) with
  | Cons(h,q) -> h
 ;;
 assert (x = 1)
```

```
x : [1,1]
```

Listing 3: `match.ml`

```
let x = match Cons(1,Cons(2, Nil)) with
  | Cons(h,q) -> h
 ;;
 assert (x < 3)
```

```
x : [1,2]
```

Listing 4: `match2.ml`

```
let x = match Cons(1,Cons(2, Nil)) with
  | Cons(h,q) -> h
 ;;
 assert (x = 1)
```

```
x : [1,2]
Warning : assertion failure
```

Listing 5: `match_alarm.ml`

L'abstraction de x est identique, mais renvoie cette fois : `Warning: assertion failure`. En effet, comme `x : [1,2]`, si `x=1`, il s'agit d'une fausse alarme - c'est effectivement le cas ici.

```
let x = match Cons(1,Cons(2, Nil)) with
  | Cons(h,q) -> h
 ;;
 assert (x = 3)
```

```
x : [1,2]
Error : assertion failure
```

Listing 6: `match_error.ml`

L'abstraction de x est identique, mais renvoie cette fois : `Error: assertion failure`. En effet, puisque l'intersection entre [1,2] et [3,3] est vide, il s'agit effectivement d'une erreur.

```
let f x y = x + y ;;
let z = f 1 2 ;;
assert (z = 3)
```

```
z : [3,3]
```

Listing 7: `add.ml`