Formalisation d'une analyse de région pour Frama-C/WP

Jérémy Damour, Allan Blanchard, Loïc Correnson, Frédéric Loulergue

Formalisation d'une analyse de région pour Frama-C/WP

Jérémy Damour[1, 2], Allan Blanchard[1], Loïc Correnson[1] et Frédéric Loulergue[2]

[1]CEA List
[2]Univ. Orléans, INSA Centre Val de Loire, LIFO EA 4022, Orléans, France

La preuve déductive en logique de Hoare de programmes avec des pointeurs nécessite une modélisation logique de la mémoire. Dans Frama-C/WP, un outil mature et utilisé industriellement pour la preuve de programmes C, nous étudions un nouveau modèle mémoire qui s'appuie sur la nature et la localisation des accès mémoires effectués par le programme. Cette cartographie des accès mémoires est une sorte analyse d'alias enrichie par des méta-données sur le type des accès effectués. Une analyse modulaire est nécessaire, avec des annotations pour en adapter la précision ainsi que des fondations sémantiques solides pour en démontrer la correction.

Cet article présente cette cartographie des accès mémoire, que nous appelons analyse de région, son implémentation, sa formalisation ainsi que sa preuve de correction sémantique. C'est une analyse d'alias et de forme, incrémentale, modulaire, qui peut être vue comme un système d'inférence de type. L'analyseur, sa formalisation et la preuve de correction sémantique sont entièrement développés dans l'assistant de preuve Coq. C'est un travail en cours de développement mais qui montre d'ores et déjà le rôle et la nécessité de conditions de vérifications spécifiques, généralement éludées dans la littérature, pour en garantir la correction.

1 Introduction

Frama-C [KPS24, BBB+21] est une plateforme logicielle *open-source* et extensible de vérification et d'analyse de programmes C, développée par le CEA-LIST. Elle propose, entre autres, un composant de preuve déductive, Frama-C/WP [BBBC24, BBC+] (*Weakest Precondition Calculus*) qui permet de prouver de manière modulaire des propriétés fonctionnelles de fonctions C exprimées dans le langage ACSL [BMP24, BFM+] au moyen de solveurs SMT [BdMS05] (*Satisfiability Modulo Theories*).

Frama-C/WP utilise une logique de Hoare [Hoa69] et transforme les instructions C ainsi que les spécifications ACSL en formules logiques du premier ordre pour générer des obligations de preuves qui seront finalement envoyées aux solveurs SMT. Au sein du processus de traduction vers cette logique du premier ordre, la modélisation de la mémoire, des pointeurs et des opérations de lecture et d'écriture joue un rôle déterminant dans la complexité des formules envoyées aux solveurs. De fait, il existe en vérification déductive différentes manières de modéliser la mémoire, ou « *modèles mémoire* ».

Nous développons actuellement un nouveau modèle mémoire pour WP qui s'appuie sur une analyse d'alias spécifique, appelée « *analyse de Région* ». Le présent article est dédié à la présentation et à la formalisation de cette analyse au sein de l'assistant de preuve Coq. Avant de présenter ce développement en détails, nous introduisons tout d'abord les liens qui existent entre Modèles Mémoire, Vérification Déductive et Analyse d'Alias.

Modèles mémoire et sémantique. Quand il s'agit de définir formellement la sémantique d'un langage, on utilise des modèles mémoire très proches de leur description informelle. Typiquement, pour le langage C, la mémoire est généralement représentée par un tableau de bits ou de d'octets indexé par les adresses mémoire. Parmi les modèles les plus aboutis pour le langage C, on trouve par exemple le modèle de CompCert [LABS12] ou celui de Krebbers [Kre12]. Ce sont des modèles très expressifs, dans le sens où ils peuvent modéliser la quasi-totalité des opérations du langage C, en donnant en particulier un sens rigoureux à la notion de *comportements bien définis*. De fait, tous les programmes C ne sont pas toujours corrects et cela se traduit par des opérations qui sont acceptées ou refusées par la sémantique du langage et tout particulierèrement par le modèle mémoire. Ainsi, les deux modèles de CompCert et de Krebbers ne définissent pas tout à fait le même ensemble d'opérations, en particulier pour les unions et les casts de pointeurs.

Modèles mémoire et vérification déductive. Bien que très expressifs, ces modèles ne sont pas bien adaptés à la vérification déductive de programmes : ils génèrent des obligations de preuve bien trop complexes que même les meilleurs solveurs SMT sont incapables de résoudre.

D'autres modèles mémoire que ceux mentionnés ci-dessus sont plus adaptés à la vérification. Par exemple, pour certains programmes, le modèle de Burstall-Bornat [Bor00] tire parti des types et de la séparation entre les champs d'une même structure de données pour partitionner la mémoire en une multitude de *régions*, chacune d'entre elles étant encodée de manière indépendante et plus simple (du point de vue des solveurs SMT) que des séquences d'octets. Bien que plus efficace pour la preuve, ces modèles mémoires sont souvent assez éloignés de la sémantique informelle du langage, ce qui pose la question de leur correction. D'ailleurs, cette correction repose souvent sur des hypothèses (plus ou moins implicites) qui font que ces modèles ne sont pas toujours applicables : il faut au préalable s'assurer que ces hypothèses sont bien vérifiées par le programme considéré et si besoin, changer de modèle bien que les preuves soient plus difficiles.

Ainsi, Frama-C/WP propose plusieurs modèles mémoire spécialement conçus pour les solveurs SMT, en proposant différents compromis entre pouvoir d'expression (applicabilité) et complexité des obligations de preuve générées. À ce jour, les modèles mémoire de WP sont sous-optimaux (vis-à-vis de la difficulté à prouver) et leurs conditions d'applications sont mal connues. Surtout, ces conditions ne sont pas toujours automatiquement contrôlées par l'outil et doivent être vérifiées par une revue manuelle de code.

Modèles mémoire, aliasing et typage. Les modèles mémoires les plus efficaces pour la preuve déductive sont ceux qui exploitent au maximum la séparation de la mémoire en zones distinctes et la connaissance des types de données lues ou écrites dans chaque zone. En C les expressions avec pointeurs peuvent potentiellement référencer plusieurs zones mémoires différentes. On parle alors d'*Aliasing* et cela entraîne la nécessité de modéliser ces zones mémoires avec des tableaux.

De plus, il est possible avec le langage C d'accéder à une variable avec un type différent de sa définition, par exemple *via* des unions ou des casts de pointeurs. Dans ce cas, il est nécessaire de conserver une représentation des données en séquence d'octets et/ou de modéliser les conversions de données d'un type de données à un autre. Cela rend les preuves plus difficiles, voire impossible si on doit le faire pour l'ensemble de la mémoire.

Vers une analyse de région. Dans le cadre du projet CoMeMoV, nous cherchons à développer un nouveau modèle mémoire *à la* Bornat pour WP basé sur une cartographie précise des accès mémoires, avec le plus d'informations possibles concernant les alias et le type des accès. Ces informations seront fournies en partie par l'utilisateur (pour avoir une analyse modulaire) et en partie inférée par une analyse automatique du programme. Cette analyse, dite « *analyse de région* » permettra à WP d'obtenir une partition de la mémoire ainsi que des informations d'aliasing et de typage afin d'optimiser au maximum les obligations de preuve générées.

Le cœur de cette analyse de région est tout à fait similaire à une analyse d'alias classique [HT01, LGRF$^+$24, DMM98, SB15, Ste96] mais elle intègre aussi des éléments d'analyse de « forme » (*Shape Analysis*) [CR08, ILR21, JLRS10a].

Par comparaison avec d'autres analyses statiques d'alias ou de forme, notre analyseur est précis pour les champs de structures [Min06] (*Field Sensitive*) et ne différencie pas les éléments de tableaux (*Array Insensitive*), ce qui permet de rester suffisament précis pour la plupart des codes C que nous considérons habituellement.

Le noyau de l'analyseur est incrémental. Il pourrait être utilisé pour effectuer une analyse globale inter-procédurale [JLRS10b, BDES11] mais WP l'utilisera plutôt de manière modulaire, à l'aide d'annotations ACSL spécifiques pour spécifier les contraintes sur les régions mémoires *via* les contrats de fonctions et ainsi aider l'analyse à être plus précise qu'une analyse globale inter-procédurale classique [Hor97].

Contributions. Dans cet article, nous nous focalisons sur l'implémentation de cette analyse de région, la formalisation des cartes de régions et la preuve de correction sémantique de l'analyseur. L'étude du modèle mémoire associé ne sera pas abordée. Ces travaux, toujours en cours de développement, sont entièrement réalisés dans l'assistant de preuve Coq et s'appuie sur une sémantique du C *générique* qui peut être instanciée à la fois sur le modèle sémantique de CompCert ou sur celui de Krebbers et plus généralement, sur la plupart des autres modèles sémantiques du C.

Les principales contributions de ce travail sont à ce jour :

- L'implémentation en Coq d'un moteur d'analyse de région incrémental produisant des informations d'aliasing, de forme et d'accès mémoires propres à être utilisées pour un modèle mémoire efficace en preuve déductive.
- La formalisation en Coq d'une carte de régions, telle que produite par notre analyseur, et son interprétation comme un système de type ou comme un domaine abstrait.
- La preuve de correction sémantique en Coq de l'analyseur, partielle pour le moment, permettant d'utiliser les cartes de régions générées comme une source d'information fiable permet de garantir des propriétés sémantiques sur les accès mémoires effectués par le programme.
- La mise en lumière de *conditions additionnelles* absolument nécessaires pour garantir cette correction sémantique, ainsi qu'une piste préliminaire pour automatiser la génération de *gardes* sous forme d'annotations ACSL permettant de s'assurer de la correction du modèle.

Ces conditions additionnelles étaient attendues bien qu'elles soient habituellement éludées ou mis de côté dans la littérature, et elles feront spécifiquement l'objet de travaux futurs.

Présentation. Après un rappel sur le langage C (section 2), nous donnons des exemples de cartes mémoires que nous pouvons générer, leur définition formelle ainsi que les notations utiles à la formalisation de leurs propriétés (section 3). Nous présentons ensuite l'algorithme d'analyse de région pour produire ces cartes que nous présentons comme un système de typage statique indépendant de la sémantique du C (section 4). Puis, nous abordons la preuve de correction sémantique de cette analyse de région (section 5).

Notations. Nous utilisons des notations classiques pour les types listes et options. Ainsi, le type $[\alpha] \triangleq [v_i]_{i \leq n}$ désigne une liste d'élements v de type α et le type $\alpha^? \triangleq \lfloor v \rfloor \mid \bot$ une valeur optionnelle de type α. On notera également *Prop* le type des propositions logiques.

2 Langage C

Nous analysons du code C, un langage de programmation impératif et de bas niveau, réputé pour sa complexité d'accès mémoire. Ainsi, nous nous intéressons spécifiquement aux affectations dans le langage C, qui manipulent directement la mémoire. En C, il est essentiel de différencier les valeurs gauche (*l-values*), qui représentent des emplacements en mémoire, des expressions évaluées produisant une valeur mais ne désignant pas nécessairement une adresse en mémoire.

$$
\begin{array}{rcrcl}
e & : & \mathit{expr} & ::= & c \\
 & & & \mid & e \odot e \\
 & & & \mid & l \\
 & & & \mid & \&\, l \\
 & & & \mid & (\tau *)\ e
\end{array}
\qquad
\begin{array}{rcrcl}
l & : & \mathit{lval} & ::= & x \\
 & & & \mid & l.f_s \\
 & & & \mid & l.f_u \\
 & & & \mid & *(e + e)
\end{array}
$$

$$
s \ : \ \mathit{stmt} \ ::= \ l = e
$$

Figure 1. Grammaire du langage C utilisée.

La grammaire présentée dans la figure 1 définit les expressions et les l-values utilisées dans le langage C. Les lectures de l-value, la prise de référence, le déréférencement et l'écriture par une affectation jouent un rôle central dans les manipulations de la mémoire. Une expression e peut représenter des valeurs littérales (c), des opérations binaires ($e \odot e$), ou des accès en lecture à des l-values (l), permettant ainsi de lire la valeur associée à un emplacement mémoire. La prise de référence ($\&\ l$) permet d'obtenir l'adresse de la l-value. De plus, des casts de pointeurs ($(\tau *)\ e$) peuvent être appliqués et changer l'interprétation d'une valeur en mémoire.

Les l-values l désignent des emplacements en mémoire, tels que des variables simples (x), l'accès à des champs de structures ($l.f_s$), ou l'accès à des champs d'unions ($l.f_u$) ce dernier étant similaire aux casts de pointeurs. Elles incluent également des accès par déréférencement et décalage de pointeurs par arithmétique de pointeurs ($*(e + e)$).

Enfin, l'instruction d'affectation, notée $l = e$, permet d'écrire une valeur, issue d'une expression e, dans une adresse contenue dans une l-value l. Cette opération modifie la mémoire et peut entraîner la création d'alias, lorsque deux l-values distinctes pointent vers la même adresse.

Ainsi, l'interaction entre lecture, prise de référence, déréférencement et affectation constituent le cœur des opérations sur la mémoire que doit capturer l'analyse de région.

3 Cartes de régions

Une carte de régions est une représentation sous forme de graphe des différents accès à la mémoire effectués par tout ou partie d'un programme au cours de son exécution. À partir des variables du programme (les racines du graphe) on peut suivre sur la carte la construction d'une l-value pour la localiser dans une *région* de la mémoire, c'est-à-dire un ensemble d'adresses bien défini.

Chaque nœud du graphe forme ainsi une région, étiquetée par des méta-données telles que la nature des accès (lecture, écriture) et les types C de toutes les l-values qui y accèdent [1]. Des

1. D'autres méta-données peuvent être collectées pour WP comme la validité ou l'initialisation.

arcs structurels entre différentes sous-régions émergent pour les types agrégats (structures, unions et tableaux), permettant de suivre les accès par champs ou par index de tableau. Enfin, les régions de la mémoire qui contiennent des adresses font apparaître des arcs « *points-to* » permettant de suivre les accès par déréférencement de pointeurs.

La définition du langage C fait que les arcs structurels forment un graphe acyclique (DAG) et permettent de donner une définition statique et bien fondée des adresses qui appartiennent à une région. Les arcs « *points-to* » en revanche, peuvent introduire des cycles.

Dans la suite de cette section, nous illustrons les cartes de régions sur deux exemples, puis nous en donnerons une définition formelle.

3.1 Exemples

Les exemples ci-dessous dans les figures 2 et 3 présentent différents codes C accompagnés de cartes de régions, générées grâce à une première implémentation de l'analyse de région dans le greffon région de Frama-C, visualisées dans l'interface graphique Ivette [Cor22].

Exemple 1 (Exemple de carte d'agrégat). La figure 2 illustre un exemple de programme C, comportant une structure avec plusieurs champs. La variable `s` est associée à une région divisée en plusieurs sous-régions, correspondantes aux différents champs de la structure. On peut observer que seuls les champs `.x` et `.z` sont accédés, laissant un espace de 32 bits vide au milieu [2]. On peut aussi remarquer que le champs `.z` est accédé comme un tableau de taille quatre par l'annotation « [4] ».

La l-value `s.x` est localisée dans une première région étiquetée « W(i) » indiquant un accès en écriture avec le type `int`, noté (i). On peut également observer la représentation structurelle du tableau `s.z` constitué d'une sous-région étiquetée aussi par « W(i) », correspondant aux accès en écriture *via* la l-value `s.z[k]`. Enfin dans la l-value `s.z[k]`, `k` est une expression lisant une valeur entière à l'adresse de `k`, ainsi étiquetée « R(i) » dans la carte de région.

D'une manière générale, si une région de taille s contient une sous-région de taille e, c'est qu'elle représente implicitement un tableau de $n = s/e$ cellules, chaque cellule ayant les même caractéristiques.

```
void f(int k) {
  struct { int x, y, z[4]; } s;
  s.x = 1;
  s.z[k] = 0;
}
```

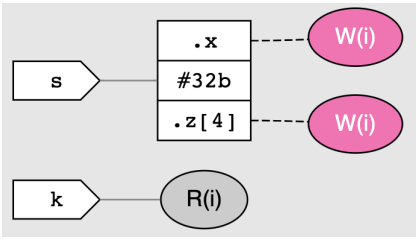

Figure 2. Exemple d'accès tableau et structure.

Exemple 2 (Exemple de carte de pointeur). La figure 3 illustre un programme C où, à cause d'un cast de pointeur, une certaine région mémoire est accédée avec des types différents de celui déclaré. Dans ce code, le pointeur `p` peut référencer soit un `short` contenu dans `x`, soit une partie d'un `int` contenue dans `y`.

La carte de régions montre que la zone mémoire associée à `p` contient bien un pointeur symbolisé par l'étiquette « RW* ». Cette région présente aussi un arc « *points-to* » vers une autre région mémoire étiquetée « W(s) » indiquant des accès en écriture avec le type `short`. Cette région est associée aux *deux* variables `x` et `y` qui sont ainsi « en alias ». De plus, on voit dans la carte de régions que la variable `y`, pourtant de type `int`, est accédée comme un tableau de deux `short`.

2. De tels espaces sont également observés en présence de *padding*.

```
void g(short x, int y) {
  short *p = &x;
  *p = 1;
  p = (short *)&y;
  p[0] = 0;
  p[1] = 1;
}
```

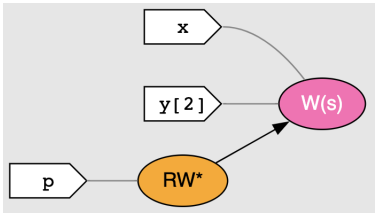

Figure 3. Exemple de pointeur avec cast.

3.2 Définition d'une carte de régions

Une carte de régions est un graphe qui peut être cyclique en raison de la présence de pointeurs. Les variables du programme (*var*) sont associées aux racines du graphe. Les nœuds du graphe sont des régions (*id*), chacune ayant une forme (*shape*). De part la nature incrémentale de l'analyse, les identifiants de région ne sont pas uniques et plusieurs identifiants peuvent en réalité être associés à la même région. Une région est donc formellement définie par une classe d'équivalence d'identifiants, au sein de laquelle tous les identifiants ont la même forme. On note ($\approx$) la relation d'équivalence associée.

Le type d'une carte de régions G est formellement défini comme suit :

$$G \ : \ (var \to id) \times (id \to_{(\approx)} shape)$$

Pour manipuler les cartes de régions, nous employons deux notations : $G[x]$ pour récupérer la région associée à une variable x, et $G[r]$ pour obtenir la forme d'une région r. Ces fonctions sont respectivement les projections du couple.

Chaque nœud avec un accès peut inclure des méta-données décrivant la région. Celles-ci regroupent plusieurs informations essentielles : la taille de la région, qui correspond au plus grand diviseur commun des tailles des types utilisés ; les types employés pour les lectures ; et ceux utilisés pour les écritures. Ces méta-données sont formalisées comme suit :

$$info \ \triangleq \ \mathbb{N} \times [\tau] \times [\tau]$$

La représentation graphique utilise ces informations pour indiquer les accès en lecture (R) et en écriture (W), ainsi qu'une abréviation du type approprié selon la taille du nœud, comme illustré dans les exemples de la section 3.1.

Les nœuds d'une région peuvent adopter différentes formes, correspondant chacune à un type d'accès spécifique à une l-value du programme :

- Blob : La région n'est pas accédée.
- Cell $\perp$ ι : La région contient uniquement des entiers primitifs, sans pointeur. Elle est annotée avec des méta-données (ι : *info*) et constitue un nœud terminal de la carte de régions.
- Cell $\lfloor r \rfloor$ ι : La région contient des entiers *ou* des pointeurs, ces derniers pointant vers la région r.
- Range $[(p_i, q_i, r_i)]_{0 \le i < n}$ ι : La région contient des agrégats (structures, unions, tableaux), décomposés en n plages d'offsets disjointes. Chaque plage $[p_i, q_i)$ est associée à une sous-région r_i, avec les contraintes $0 \le p_0$, $p_i < q_i \le p_{i+1}$, et $q_{n-1} \le s$ (taille de l'agrégat). Ces contraintes garantissent que les plages sont disjointes et qu'elles restent dans les limites de l'agrégat.

La forme des nœuds est formellement définie par le type suivant :

$$\begin{aligned} shape \ \triangleq \ & \text{Blob} \\ \mid \ & \text{Cell } (r : id^?) \ (\iota : info) \\ \mid \ & \text{Range } (\rho : [\mathbb{N} \times \mathbb{N} \times id]) \ (\iota : info) \end{aligned}$$

Pour implémenter cette carte de régions, nous adoptons une structure d'*Union-Find* [CP19]. Cette structure est particulièrement efficace pour fusionner rapidement des régions au sein de la carte de régions. L'axiomatisation de cette structure [3] ainsi que la définition de la carte de régions [4] est accessible dans le développement Coq.

3.3 Notations

Nous adoptons les notations de la figure 4 pour désigner les relations dans la carte de régions. Ces notations établissent un cadre précis pour décrire la structure et les relations entre les différentes régions de mémoire. Elles sont définies de manière rigoureuse dans les sections 5.1 et 5.2, servant ainsi de fondement à nos développements théoriques et formels. De plus, ces notations sont également présentes dans le développement Coq [5].

(1)	$G \vdash x \mapsto r$	La variable x est affectée à la région r.
(2)	$G \vdash r_1 \hookrightarrow r_2$	La région r_1 pointe vers la région r_2.
(3)	$G \vdash r_1 \xmapsto{p..q} r_2$	La région r_1 contient une sous-région r_2 pour le segment $[p, q)$.
(4)	$G \vdash r_1 \xmapsto{.f} r_2$	La région r_1 contient une sous-région r_2 pour le champ $.f$.
(5)	$G \vdash r_1 \overset{*}{\mapsto} r_2$	La région r_1 contient une sous-région r_2 (transitivement).
(6)	$G \vdash r \mid \tau$	La région r a une taille qui divise celle du type τ.
(8)	$G \vdash l : r$	La l-value l est localisée dans la région r.
(8)	$G \vdash e : d$	L'expression e a pour type (ou domaine) d.
(9)	$G \vdash a \in r$	L'adresse a appartient à l'empreinte de la région r.
(9)	$G \vdash v \in d$	La valeur concrète v appartient au domaine (ou type) d.
(10)	$G \vdash r_1 \# r_2$	Les régions r_1 et r_2 sont séparées.

Figure 4. Notations utilisées pour les cartes de régions (Définies dans sections 5.1 et 5.2).

Exemple 3 (Exemple d'utilisation de notation). La figure 5 présente un exemple de programme C ainsi que la représentation graphique associée de la carte de régions, que l'on notera G. Cette dernière sert de support pour illustrer les différentes notations introduites dans la figure 4.

- $G \vdash s \mapsto r_s$: La variable `s` est associée à la région r_s dans la carte de régions G. De même, la variable `k` est associée à la région r_k, et `p` à la région r_p.
- $G \vdash r_p \hookrightarrow r_f$: La région r_p pointe vers la région r_f.
- $G \vdash r_s \xmapsto{.f} r_f$: La région r_s contient une sous-région r_f, définie par les *offsets* du champ `.f` de la structure `s`.
- $G \vdash r_s \xmapsto{p..q} r_f$: La région r_s contient une sous-région r_f, définie par les *offsets* du champ `.f`, comme mentionné précédemment. Ce segment $[p, q)$ peut aussi correspondre à un segment valide pour l'accès au tableau contenu dans ce champ.
- $G \vdash r_f \mid$ `int` : La région r_f est alignée pour contenir des valeurs de type `int` (peut accueillir jusqu'à cinq entiers).
- $G \vdash$ `s.f[k]` $: r_f$: La l-value `s.f[k]` est localisée dans la région r_f.
- $G \vdash$ `p` : Pointer r_p : L'expression de lecture de la l-value (de type pointeur) `p` est typée dans une région de pointeurs r_p.
- $G \vdash$ `*(p + k)` : Primitive : L'expression de déréférencement de `p` avec un décalage de `k` est une valeur primitive appartenant au domaine des valeurs primitives (Primitive). De même pour l'expression `k` ou encore `s.f[k]`.
- $G \vdash a_f \oplus i \in r_f$ avec a_f l'adresse de la l-value $s.f$: Les adresses calculées à partir de a_f et d'un décalage i (tel que $0 \leq i < 5 \times$ sizeof(`int`)) appartiennent à la région r_f.

3. Cf. fichier `theories/Utils/UnionFind.v`
4. Cf. fichier `theories/RegionMap/Core.v`
5. Cf. fichier `theories/RegionMap/Relations.v`

— $G \vdash c \in \mathsf{Primitive}$: Toute valeur primitive c appartient au domaine des valeurs primitives (Primitive).
— $G \vdash r_p \# r_f$: La région r_p est séparée (ou disjointes) de la région r_f. De même, la région r_p est séparée de r_s. Et r_p, r_s et r_f sont séparées de r_k.
— $G \vdash r_s \overset{*}{\mapsto} r_f$: la région r_f est une sous-région de r_s (donc non disjointes).

```
void h(struct S s, int k) {
  int *p = s.f;
  p[k] = 0;
}
```

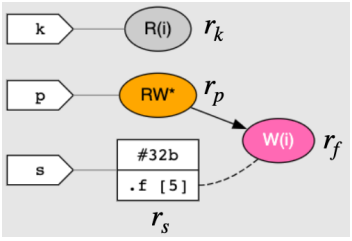

Figure 5. Exemple avec plusieurs accès.

3.4 Monade

Pour simplifier l'utilisation des fonctions associées à la carte de régions, nous introduisons une monade [SBGG13] d'état et d'erreur. Cela permet de structurer les opérations complexes de manipulation des régions en encapsulant l'état mutable résultant de la structure Union-Find. Ainsi, des opérations telles que la création, la recherche, la fusion de régions ou l'ajout d'accès peuvent être chaînés de manière claire et concise, ce qui renforce la lisibilité et l'écriture du code. Cette monade est définie de par les types suivants :

$$\begin{array}{lcl} \mathit{result} & \triangleq & \forall\ \sigma\ \alpha,\quad \mathsf{Return}\ (s : \sigma)\ (r : \alpha)\ |\ \mathsf{Error} \\ S & \triangleq & \forall\ \sigma\ \alpha,\quad \sigma \to \mathit{result}\ \sigma\ \alpha \end{array}$$

Nous définissons des fonctions pour manipuler la monade, intégrant état et erreur. Les opérations bind et ret permettent de composer et d'encapsuler des valeurs, facilitant le chaînage. Nous étendons cela avec set et get, qui modifient et lisent l'état interne, tandis que fail gère les erreurs en encapsulant les exceptions dans la monade.

Nous définissons des notations pour faciliter la manipulation de la monade :
— `let* x = e1 in e2` : Exprime l'opération monadique bind pour enchaîner les opérations.
— `e1 ;; e2` : Représente une séquence d'opérations où `e1` est exécutée avant `e2`.

4 Analyse de région

L'analyse de région produit une carte de régions à partir d'un programme C donné. Cette carte permet de localiser les l-values, d'enregistrer les différents accès effectués et de déterminer quelles l-values sont disjointes en mémoire. Cette analyse est un algorithme additif, c'est-à-dire qu'il peut incorporer des informations provenant de différents points de programme, indépendamment de leur ordre d'exécution. On peut aussi l'utiliser pour tout ou une partie du programme, ce qui permet d'obtenir une analyse inter-procédurale ou modulaire. Le cœur de l'analyse se concentre principalement sur les l-values (l), les expressions (e) et les instructions d'affectations ($l = e$), qui entraînent des accès en mémoire ou la création d'alias.

Actuellement, l'analyse ne prend pas en charge les appels de fonctions ainsi que les copies de structures ou d'unions. Ces constructions feront l'objet d'extensions et de travaux futurs.

Pour plus de concision dans la rédaction de l'algorithme, nous utilisons une monade d'état et d'erreur (voir la section 3.4). L'implémentation de l'analyse de région est réalisée en Coq [6].

6. Voir le fichier `theories/RegionAnalysis/Core.v`

4.1 Typage et domaine de valeurs

Cette analyse fonctionne de manière similaire à une inférence de type ou à une analyse de valeur par interprétation abstraite [CC77]. En effet, elle construit une carte de régions G qui permet localiser les l-values l dans la région r, noté $G \vdash l : r$.

De même, quand on considère une expression e dans la carte G, e ne prend que des valeurs primitives ; ou bien e peut avoir la valeur d'un pointeur vers la région r.

Pour formaliser cette approche, nous introduisons le type des domaines de valeurs :

$$\mathit{dval} \triangleq \mathsf{Primitive} \mid \mathsf{Pointer}\ (r : \mathit{id})$$

On pourra ainsi dire que, dans la carte (l'environnement) G, une expression e a le type $d : \mathit{dval}$ ou que e prends ses valeurs dans le domaine d. On notera ainsi $G \vdash e : \mathsf{Primitive}$ le cas où e ne prend que des valeurs primitives, et $G \vdash e : \mathsf{Pointer}\ r$ le cas où e peut s'évaluer en des adresses qui pointent vers la région r.

4.2 Fusion de régions

Pour implémenter notre analyse incrémentale de carte de régions, une fonction de fusion est nécessaire. Elle permet de combiner deux régions tout en préservant leurs méta-données respectives. Intuitivement, la fusion de deux régions mémoires réunit ces deux régions en une seule qui contiendra toutes les adresses des deux régions initiales, ainsi que toute information sur les accès mémoire qui y sont associés.

Elle assure ainsi la divisibilité des tailles entre les régions, ce qui permet la création de tableaux implicites. De plus, elle doit calculer une forme commune compatible avec les deux formes initiales, ce qui peut conduire à des fusions en cascade ou à la réunion de plusieurs segments d'offsets en un unique segment pour les agrégats.

Pour des structures de même taille, la fonction de fusion permet ainsi de juxtaposer des segments d'agrégats disjoints. En cas de chevauchement de deux segments, il faut alors fusionner les deux segments et les sous-régions associées tout en vérifiant les contraintes d'alignement.

L'expression de cette fonction de fusion est difficile en Coq, car elle nécessite un critère de terminaison qui indique la fin des fusions successives. En raison de cette complexité, elle est pour le moment axiomatisée dans notre développement Coq. Cependant, une version de référence de cette opération est implémentée en OCaml dans Frama-C/WP et servira de base à sa formalisation en Coq par la suite.

4.3 Analyse des expressions et des l-values

L'analyse des expressions est effectuée par la fonction `analyse_expr`. Cette fonction modifie la carte de régions pour localiser l'expression et renvoie le domaine abstrait correspondant. Elle renvoie Primitive si l'expression est primitive, et Pointer r si l'expression est de type pointeur. L'analyse d'une l-value est réalisée par la fonction `analyse_lval` mutuellement récursive avec `analyse_expr`. Cette fonction modifie la carte de régions afin d'exprimer la l-value analysée et renvoie la région correspondante à cette l-value.

La figure 6 illustre d'abord l'analyse d'une expression issue de la lecture d'une l-value `l`. On commence par analyser la l-value, ce qui modifie la carte de régions et renvoie la région `reg` associée à cette expression. Ensuite, on ajoute l'accès en écriture à cette région avec son type τ.

Si l'expression est de type primitif, elle est localisée dans le domaine Primitive. Sinon l'expression est de type pointeur. On commence par créer deux régions fraîches, `src` et `tgt`, avec une relation « *points-to* » entre elles. On fusionne la région `reg` dans laquelle est localisée la l-value avec la source du pointeur `src` et renvoie le domaine Pointer `tgt`.

La figure 6 montre aussi l'analyse d'une l-value accédant à un champ `f` d'une l-value (de type structure) `s`. Pour ce faire, nous commençons par analyser la l-value `s`, ce qui nous

fournit une région `r`. Nous calculons ensuite les offsets `p` et `q` du champ `f`. Nous créons alors deux nœuds, `comp` et `sub`, où la région `comp` a une taille correspondant à celle de la structure, donnée par son type τ, et inclut une sous-région `sub` pour le segment `p` à `q`. Enfin, nous fusionnons la région `r` de la l-value précédemment analysée avec la région `comp` et renvoyons la sous-région `sub`, qui contient le champ `f`.

À noter que cette opération n'est réalisée *que* pour les champs de structures. Pour les champs d'union, on ne créé pas de sous-région, mais on se contente de renvoyer la région initiale. Ce traitement des unions est de fait le même que pour les casts de pointeurs, qui ne changent pas la région mémoire accédée.

```
Fixpoint analyse_expr (e : expr) : S G dval :=
  match e with
  | E_lval l ⇒
    let* r := analyse_lval l in          (* Analyse de la l-value l *)
    let* τ := typeof e in
    read r τ;;                           (* Ajoute l'accès en lecture avec le type de e *)
    if (is_ptrb τ) then (
      let* (src, tgt) := new_pointer in  (* Créé deux régions liées par ⊢ src ↪ tgt *)
      merge r src;;                      (* Fusion des régions pointeurs r et src *)
      ret (Pointer tgt)                  (* Renvoie la région pointée tgt *)
    ) else (
      ret Primitive)                     (* Renvoie une région primitive *)
  ...
  end with analyse_lval (l : lval) : S G id :=
  match l with
  | L_struct s f ⇒
    let* r := analyse_lval s in          (* Analyse de la structure s *)
    let* τ := typeof s in
    let* (p, q) := offset_of τ f in      (* Calcule l'offset du champ f *)
    let* (comp, sub) := new_range τ p q in (* Créé deux régions liées par ⊢ comp -.f-> sub *)
    merge r comp;;                       (* Fusionne les régions r et comp *)
    ret sub                              (* Renvoie la région du champ f *)
  ...
  end.
```

Figure 6. Exemple d'analyse d'une expression ou d'une l-value.

4.4 Analyse de l'affectation

L'analyse d'une instruction comme une affectation se fait par la fonction `analyse_stmt`. Cette fonction modifie la carte de régions en ajoutant les différentes expressions et l-values analysées ainsi que des arcs « *points-to* » en cas d'alias.

La figure 7 illustre l'analyse d'une affectation. Pour cela, on commence par analyser la l-value `l`, ce qui nous donne une région `r` dans laquelle est localisée cette l-value. On ajoute ensuite l'écriture dans cette l-value avec son type τ. On analyse ensuite l'expression `e`. Si celle-ci est primitive, il n'y a rien à faire. En revanche, si elle est de type pointeur et contient une région pointée `p`, alors on créé deux nouvelles régions `src` pointant vers `tgt`. Ensuite, on fusionne la source du pointeur `src` avec la région `r` de la l-value `l`, et la cible du pointeur `tgt` avec la région pointée `p`.

```
Definition analyse_stmt (s: stmt) : S G unit :=
  match s with
  | S_assign l e ⇒
      let* r := analyse_lval l in            (* Analyse la l-value l *)
      let* τ := typeof l in
      write r τ;;                            (* Ajoute un accès en écriture à r *)
      match* analyse_expr e with
      | Primitive ⇒ ret tt                   (* La région est primitive *)
      | Pointer p ⇒                          (* Si e est un pointeur localisé dans p *)
          let* (src, tgt) := new_pointer in  (* Créé deux régions liées par ⊢ src ↪ tgt *)
          merge r src;;                      (* Fusionne les régions pointeurs r et src *)
          merge p tgt                        (* Fusionne les régions pointées p et tgt *)
      end
  end.
```

Figure 7. Analyse de l'affectation.

5 Correction

La fonction principale d'une carte de régions est de localiser les l-values d'un programme C. La relation de localisation des l-values et typage des expressions par un domaine de valeur (définition 8) joue un rôle central en garantissant que chaque adresse d'une l-value ou valeur d'une expression est correctement identifiée dans la carte de régions. Cette localisation repose sur des relations primitives entre régions, telles que la relation d'étiquetage (définition 1), la relation *points-to* (définition 2) et les relations d'agrégation (définitions 3, 4 et 5). Cette définition inductive doit être vérifiée pour toutes les sous-expressions qui composent l'expression typée, ce qui est l'objet du lemme 1.

Pour vérifier la validité de notre analyse, nous souhaitons prouver que celle-ci conserve toutes les localisations des l-values précédentes (lemme 2) et qu'elle ajoute correctement la localisation de la l-value analysée (lemme 3).

Ces définitions sont statiques et indépendantes de toute sémantique du C. Si nous pouvons typer une expression de manière statique, alors nous pouvons également localiser ses valeurs dans un domaine abstrait (définition 9). Ce qui justifie la présentation des cartes de régions comme des environnements de typage statique.

On peut également définir la séparation ou l'inclusion des régions de manière statique (définition 10) en introduisant l'ensemble des adresses associées à une région. Il se trouve que l'ensemble des adresses associées à une région ne dépend pas des arcs « *points-to* » et peut donc être déterminé de manière purement statique. Finalement, quand on considère deux régions, on peut démontrer que soit elles sont disjointes, soit l'une est inclue dans l'autre (théorème 1).

Néanmoins, pour faire le lien entre les cartes de régions et la sémantique du C, il est essentiel d'assurer la cohérence de la carte de régions vis-à-vis de la mémoire. Pour ce faire, nous définissons la notion de cohérence de la mémoire (définition 11) et prouvons que cette cohérence permet de donner un sens à la notion de typage par un domaine de valeur par rapport aux valeurs possibles à l'exécution, en conformité avec la sémantique du C (lemme 4). Ensuite, nous vérifions la correction de l'analyse en nous assurant qu'elle conserve la cohérence mémoire après chaque analyse d'affectation (lemme 5) et après une analyse d'un ensemble d'affectations (lemme 6).

De plus, nous vérifions que chaque valeur dans la sémantique du C est bien représentée dans la sémantique des régions (théorème 2). Nous pouvons alors conclure que la carte mémoire obtenue est correcte en tout point du programme (ou de la partie de programme) analysé (théorème 3).

5.1 Correction statique

Dans cette section, nous définissons formellement les relations de la figure 4 qui permettent d'exprimer les propriétés statiques d'une carte de régions, et en font ainsi un système de typage. Nous prouvons des propriétés intrinsèques à ces définitions, comme la clôture par sous-expression ou la monotonie de l'analyse d'expression, ainsi que la correction interne de l'analyse elle-même : si on analyse une expression e, on obtient un domaine d qui est bien le domaine de valeur de e dans la carte de régions.

Définition 1 (Relation d'étiquetage)**.** $G \vdash x \mapsto r$ représente la relation selon laquelle la variable x est associée à la région r. Cette relation est définie comme suit :

$$G \vdash x \mapsto r \triangleq G[x] = \lfloor r' \rfloor \quad avec \quad r \approx r'$$

Définition 2 (Relation Points-to)**.** $G \vdash r_1 \hookrightarrow r_2$ est une relation qui exprime que tout pointeur dans la région r_1, dans la carte de régions G, pointe vers la région r_2. Formellement, cette relation est donc définie comme suit :

$$G \vdash r_1 \hookrightarrow r_2 \triangleq G[r_1] = \mathsf{Cell}\ \lfloor r_2' \rfloor\ \iota \quad avec \quad r_2 \approx r_2'$$

Définition 3 (Relation de sous-région)**.** $G \vdash r_1 \overset{p..q}{\longmapsto} r_2$ exprime que la région r_2 est une sous-partie de la région r_1, délimitée par les indices p et q. Cela signifie que la forme de région r_1 contient un sous-segment qui inclut le segment $[p, q)$:

$$\begin{aligned} G \vdash r_1 \overset{p..q}{\longmapsto} r_2 \quad &\triangleq \quad G[r_1] = \mathsf{Range}\ [\ldots (p_i, q_i, r_i) \ldots]\ \iota \\ &avec \quad p_i \leq p < q \leq q_i,\ r_i \approx r_2 \end{aligned}$$

Cette définition de la relation de sous-région permet de démontrer que la propriété reste invariante, même après une fusion de régions (section 4.2).

Définition 4 (Sous-région de champs)**.** $G \vdash r_1 \overset{.f}{\longmapsto} r_2$ est la relation $G \vdash r_1 \overset{p..q}{\longmapsto} r_2$ avec p et q la plage d'offsets couverte par le champs f.

Définition 5 (Clôture transititve de sous-région)**.** $G \vdash r_1 \overset{*}{\mapsto} r_2$ représente la clôture transitive de la relation $G \vdash r_1 \overset{p..q}{\longmapsto} r_2$.

Définition 6 (Taille de région)**.** $G \vdash r \mid \tau$ exprime que la région r a une taille qui divise la taille du type τ. Pour les régions non-accédées (de forme Blob) la taille associée peut être considérée comme nulle, compatible avec tout type.

Définition 7 (Décalage valide)**.** $G \vdash \delta \in \mathsf{offsets}\ r\ s$ vérifie si un décalage δ est valide pour une région r, en fonction d'une taille s spécifiée. Elle permet de calculer les empreintes mémoires des différentes cellules d'un tableau :

$$\begin{aligned} G \vdash \delta \in \mathsf{offsets}\ r\ s \quad &\triangleq \quad G[r] = \mathsf{Blob} \ \vee \delta \in \left\{ k \times s_r \mid k \in \mathbb{N}, 0 \leq k < \frac{s}{s_r} \right\} \\ &avec \quad G \vdash \mathsf{sizeof}\ r = \lfloor s_r \rfloor \end{aligned}$$

Définition 8 (Localisation)**.** $G \vdash e : d$ est une relation inductive sur les expression, indiquant qu'une expression e est bien typée dans une valeur abstraite d. Et $G \vdash l : r$ est une relation sur les l-values, indiquant qu'une l-value l est localisée dans une région r selon la sémantique d'une carte de régions G. Ces deux relations sont mutuellement récursives.

Voici quelques règles extraites de la définition inductive de la localisation en Coq [7] :

— **Addr** : Si une l-value l est localisée dans la région r, alors prendre la référence de cette l-value est une expression typée par la valeur abstraite $\mathsf{Pointer}\ r$.

7. Dans le fichier `theories/RegionMap/Relations.v`

— **Expr-Lval-Primitive** : Si une l-value de type primitif σ est localisée dans une région r avec accès en lecture, alors l'expression de lecture de la l-value l est localisée dans une valeur abstraite Primitive.
— **Expr-Lval-Pointer** : Si une l-value l est de type pointeur $\tau*$ localisée dans la région r et pointant vers r' et que cette région r est lue, alors l'expression de lecture de la l-value l est typée par la valeur abstraite Pointer r'.
— **Lval-Struct** : Si la l-value s (de type structure) est localisée dans une région r avec une sous-région r' sur le segment du champ $.f$ et que la taille de la sous-région r' est compatible avec le type de la l-value $s.f$, alors la l-value $s.f$ est localisée dans cette sous-région r'.

Formellement, ces règles de localisation sont définies dans la figure 8.

$$\frac{G \vdash l : r}{G \vdash \& \; l : \mathsf{Pointer}\ r}\ \text{Addr}$$

$$\frac{\mathsf{typeof}(l) = \sigma \qquad G \vdash l : r \qquad G \vdash r : \mathsf{read}\ \sigma}{G \vdash l : \mathsf{Primitive}}\ \text{Expr-Lval-Primitive}$$

$$\frac{\mathsf{typeof}(l) = \tau* \qquad \begin{array}{c} G \vdash r \hookrightarrow r' \\ G \vdash l : r \end{array} \qquad G \vdash r : \mathsf{read}\ \tau*}{G \vdash l : \mathsf{Pointer}\ r'}\ \text{Expr-Lval-Pointer}$$

$$\frac{\mathsf{typeof}(s.f) = \tau \qquad G \vdash r \xrightarrow{.f} r' \qquad G \vdash s : r \qquad G \vdash r' \mid \tau}{G \vdash s.f : r'}\ \text{Lval-Struct}$$

Figure 8. Relation de localisation

Lemme 1 (Typage des sous-expression)**.** *Pour tout e' une sous-expression de e. Si e est typée par une valeur abstraite d dans une carte de régions G ($G \vdash e : d$), alors il existe une valeur abstraite d' où la sous-expression e' est également localisée.*

$$e' \sqsubseteq e \;\rightarrow\; G \vdash e : d \;\rightarrow\; \exists\, d',\; G \vdash e' : d'$$

Démonstration. La preuve de ce lemme se fait par induction sur la structure des sous-expressions de e. Si une expression e est correctement localisée, chaque sous-expression e_i hérite de cette relation de localisation. □

Cette preuve est formalisée en Coq [8]

Lemme 2 (Analyse monotone d'une l-value)**.** *Soit une l-value l localisée dans une région r dans la carte des régions initiale G. Si l'analyse d'une autre l-value l' à partir de cette carte G produit une carte finale G' et renvoie la région r', alors après l'analyse, la l-value l est toujours localisée dans la région correspondant à la carte finale G'. Un lemme similaire existe pour les expression et est mutuellement récursif avec ce lemme. Formellement :*

$$G \vdash l : r \rightarrow \mathsf{analyse\_lval}\ l'\ G = \mathsf{Return}\ d\ G' \rightarrow G' \vdash l : r$$

Démonstration. Cette preuve se fait par induction sur la l-value l' et utilise des preuves de monotonie sur les différentes fonctions impliquées dans l'analyse. Ces preuves de monotonie reposent elles-mêmes sur d'autres preuves similaires appliquées aux relations définissant la localisation des l-value et le typage des expression par un domaine de valeur. □

8. Cf. lemme `subexpr_is_typed_reg` contenu dans le fichier `theories/RegionMap/In.v`

Cette preuve a été mécanisée en Coq [9] en utilisant des type classes [SvdW10], qui offrent un moyen concis d'exprimer un ensemble d'hypothèses. Ces hypothèses permettent de prouver que plusieurs relations sont monotones, indépendamment de la fonction considérée. Pour chaque fonction, il est possible de définir une instance de cette type class, ce qui permet de bénéficier automatiquement de tous les lemmes relatifs à la monotonie, sans avoir à les redémontrer individuellement.

Lemme 3 (Correction de l'analyse d'une expression)**.** *Supposons que l'analyse d'une expression e à partir d'une carte de régions initiale G renvoie une valeur abstraite d ainsi qu'une carte de régions finale G'. Après cette analyse, la carte de régions finale G' localise désormais l'expression e dans la valeur abstraite d. Ce lemme est mutuellement récursif avec un lemme similaire sur les l-values. Formellement :*

$$\mathsf{analyse\_expr}\ e\ G = \mathsf{Return}\ d\ G' \to G' \vdash e : d$$

Démonstration. La preuve repose sur une induction sur l'expression e, utilisant les propriétés de monotonie de l'analyse mentionnées précédemment, ainsi que sur la correction des différentes fonctions qui interviennent dans l'analyse. □

Cette preuve est formalisée en Coq [10].

5.2 Séparation

Dans cette section, nous nous focalisons sur l'utilisation des cartes de régions en tant qu'analyse d'alias. Plus précisément, nous définissons l'empreinte mémoire d'une région, c'est-à-dire l'ensemble des adresses qu'elle représente. Cette empreinte est définie purement de manière statique. Nous démontrons enfin qu'étant données les empreintes de deux régions, soit l'une est inclue dans l'autre, soit elles sont disjointes.

Définition 9 (Empreinte Mémoire)**.** $G \vdash a \in r$ est le prédicat inductif exprimant qu'une adresse a appartient à l'empreinte mémoire de la région r. Il est définit par les règles (exhaustives) suivantes :

- **Addr-Var** : Si la variable x est localisée dans la région r, et que l'offset δ est valide pour cette région, alors l'adresse de la variable x avec un décalage δ appartient à r.
- **Addr-Compound** : Si r' est une sous-région de r sur le segment $[p, q)$ et que l'offset δ est valide pour r' et que l'adresse a appartient à r, alors l'adresse avec les décalages p et δ appartient à r'.

Formellement cela se traduit par les règles d'inférences de la figure 9.

$$\frac{G \vdash x \mapsto r \qquad G \vdash \delta \in \mathsf{offsets}\ r\ (\mathsf{sizeof}(x))}{G \vdash (\mathsf{baseof}(x) \oplus \delta) \in r}\ \text{Addr-Var}$$

$$\frac{G \vdash r \xrightarrow{p..q} r' \qquad G \vdash \delta \in \mathsf{offsets}\ r'\ (q - p) \qquad G \vdash a \in r}{G \vdash (a \oplus p + \delta) \in r'}\ \text{Addr-Compound}$$

Figure 9. Empreinte Mémoire

9. En s'appuyant sur l'instance `MonotonyAnalyseLval` de la *type class* `Monotony`, définie dans le fichier `theories/RegionAnalysis/Monotony.v`, et sur le lemme `monotony_localised`, présent dans le fichier `theories/RegionMap/Monotony.v`.

10. Grâce au lemme `conservation_expr_lval` présent dans le fichier `theories/RegionAnalysis/Conservation.v`

On peut étendre naturellement la définition des empreintes mémoire aux domaines de valeurs de la manière suivante :

$$\begin{array}{rcl} v \in \mathbb{Z} & \iff & G \vdash v \in \mathsf{Primitive} \\ (v = a \wedge G \vdash a \in r) & \iff & G \vdash v \in \mathsf{Pointer}\ r \end{array}$$

Définition 10 (Séparation)**.** $G \vdash r_1 \# r_2$ indique que les régions r_1 et r_2 dans la carte G sont disjointes, c'est-à-dire que leurs plages d'adresses ne se chevauchent pas. Formellement :

$$\begin{array}{rcl} G \vdash r_1 \# r_2 & \triangleq & \forall\ a_{1,2},\ G \vdash a_1 \in r_1 \to G \vdash a_2 \in r_2 \to (a_1 \oplus 0..s_1) \cap (a_2 \oplus 0..s_2) = \emptyset \\ \textit{avec} & & G \vdash \mathsf{sizeof}\ r_i = \lfloor s_i \rfloor \end{array}$$

Théorème 1 (Théorème de séparation)**.** *Soient deux régions r_1 et r_2 appartenant à une carte de régions G. Ces régions sont soit contenues l'une dans l'autre $G \vdash r_i \overset{*}{\mapsto} r_j$, ou elles sont totalement disjointes $G \vdash r_1 \# r_2$ Formellement, cela s'exprime par :*

$$G \vdash r_1 \overset{*}{\mapsto} r_2 \quad \vee \quad G \vdash r_2 \overset{*}{\mapsto} r_1 \quad \vee \quad G \vdash r_1 \# r_2.$$

Démonstration. Nous savons que $G \vdash r_i \overset{*}{\mapsto} r_j$, et que le chemin est contenu dans un graphe acyclique de régions (sans passer par aucun pointeur). Dans ce graphe acyclique de régions, quatre cas sont possibles :

— Si l'une des deux régions appartient au sous-graphe issu de l'autre région, alors on peut conclure directement par $G \vdash r_1 \overset{*}{\mapsto} r_2$ ou $G \vdash r_2 \overset{*}{\mapsto} r_1$.

— Si les régions r_1 et r_2 appartiennent à deux parties non-connexes du graphe, on peut montrer que leurs adresses de bases (leurs racines) sont distinctes en utilisant les règles *Addr-Var* et *Addr-Compound* de la définition 9. On conclut alors que les deux régions sont disjointes.

— Sinon, les régions r_1 et r_2 partagent un ancêtre commun, qui est une région composée. Si les deux régions ne se trouvent pas dans le même segment, la règle *Addr-Compound* permet de conclure qu'elles sont séparées. Sinon, elles sont égales.

□

Ce théorème constitue l'objectif principal de notre analyse de région : il permet la construction d'un modèle mémoire à régions. Ce modèle est essentiel pour démontrer des propriétés de programmes liées à la mémoire. Cependant, ce théorème n'est pas encore formalisé en Coq.

5.3 Sémantique du C

Nous nous intéressons ici aux affectations dans le langage C, qui opèrent directement sur la mémoire. Comme mentionné dans la section 2, la sémantique des l-values consiste à calculer une adresse dans une mémoire donnée, tandis que la sémantique des expressions consiste à produire des valeurs. Enfin, les instructions telles que les affectations modifient la mémoire en fonction des adresses et des valeurs calculées :

$$[\![l]\!]^{\star}_m : addr^? \quad [\![e]\!]_m : value^? \quad [\![l = e]\!]_m : mem^?$$

Cette sémantique repose sur un modèle mémoire capable de fournir des fonctions spécifiques pour la manipulation de la mémoire. Notre approche est compatible avec n'importe quel modèle mémoire pour le C, tels que ceux de Krebbers ou CompCert à condition qu'il implémente les fonctions suivantes :

— **Lecture en mémoire** : $\mathsf{load}\ m\ a\ \tau = \lfloor v \rfloor$. Cette fonction charge la valeur v depuis la mémoire m à l'adresse a, en respectant la taille correspondant au type τ, si l'accès correspond à une zone allouée.

— **Écriture en mémoire** : $\mathsf{store}\ m_0\ a\ v = \lfloor m \rfloor$. Cette fonction stocke la valeur v à l'adresse a dans la mémoire, en prenant une mémoire initiale m_0, et renvoie une nouvelle mémoire m après une écriture réussie.
— **Décalage par indice** : $\mathsf{shift}\ m\ a\ k\ \tau = \lfloor a' \rfloor$. Cette fonction calcule une nouvelle adresse a', obtenue en décalant l'adresse de base a de k unités dans la mémoire, en tenant compte de la taille du type τ, si cela est faisable.
— **Décalage par champ** : $\mathsf{shift_f}\ m\ a\ f = \lfloor a' \rfloor$. Cette fonction renvoie l'adresse a' correspondant à un champ f d'une structure ou d'une union, en partant de l'adresse de base a dans la mémoire m.

Cette définition de la sémantique du C est compatible avec les modèles mémoires de CompCert et de Krebbers, même si ces derniers ne sont pas nécessairement en accord avec les cas où la sémantique des différentes opérations est définie. Typiquement, les règles de strict-aliasing et de conversion de pointeurs sont plus strictes dans le modèle de Krebbers que dans le modèle de CompCert. De même la notion d'adresse n'est pas identique dans les deux modèles. Cependant, on peut dans chacun de ces deux modèles sémantiques définir une relation d'équivalence entre adresses en s'appuyant sur des notions de base et d'offset qui soient compatibles avec nos définitions.

5.4 Correction dynamique

Dans cette section, nous nous intéressons à la correction sémantique des cartes de régions, c'est-à-dire les relations entre, d'une part les propriétés statiques des cartes (le typage) que nous avons vu dans les sections précédentes et, d'autre part, les valeurs calculées en tout point de programme conformément à la sémantique du C.

La notion clef qui permet de faire cette relation entre typage statique et exécution dynamique est celle de cohérence entre un état mémoire m et une carte de régions G. Cette notion exprime tout simplement le fait que les valeurs et, plus particulièrement, les adresses que l'on pourra lire en mémoire depuis une région r vont pointer vers une région r' telle que $G \vdash r \hookrightarrow r'$. À partir de cette notion, on pourra énoncer et démontrer les théorèmes de correction sémantique de l'analyse de région.

Définition 11 (Mémoire cohérente)**.** $G \vdash m$ indique qu'une carte de régions G est cohérente avec une mémoire m. Cela signifie que si une adresse a_1, à laquelle est stockée une adresse a_2 dans la mémoire m, alors lorsque la région r_1 pointe vers la région r_2 et que l'adresse a_1 appartient à r_1, alors l'adresse a_2 appartient à r_2. Formellement :

$$G \vdash m \triangleq \forall\ a_{1,2}\ r_{1,2}\ \tau,\quad \mathsf{load}\ m\ a_1\ \tau = \lfloor a_2 \rfloor \to\ G \vdash r_1 \hookrightarrow r_2 \to\ G \vdash a_1 \in r_1 \to\ G \vdash a_2 \in r_2$$

Lemme 4 (Cohérence mémoire-région)**.** *Pour toute carte de régions G cohérente avec une mémoire m, si l'interprétation d'une l-value renvoie une adresse a_1 et que l'évaluation de cette l-value en tant qu'expression renvoie une adresse a_2, et si la région r_1 pointe vers la région r_2, alors l'adresse a_2 appartient bien à la région r_2.*

$$G \vdash m \to\ [\![l]\!]^\star_m = \lfloor a_1 \rfloor \to\ [\![l]\!]_m = \lfloor a_2 \rfloor \to\ G \vdash r_1 \hookrightarrow r_2 \to\ G \vdash a_1 \in r_1 \to\ G \vdash a_2 \in r_2$$

Démonstration. La preuve de ce lemme repose sur la sémantique des expressions et des l-values. Lors d'une évaluation d'une lecture d'une l-value (étant une expression), la sémantique charge la valeur à l'adresse de la l-value. Ici l'adresse de la l-value l est a_1 et sa valeur est une nouvelle adresse a_2.

La cohérence entre la mémoire et la carte de régions garantit que, si a_1 est dans la région r_1 et que r_1 pointe vers r_2, alors a_2 se trouve bien dans r_2. Ce raisonnement utilise la relation de chargement en mémoire et la propriété de cohérence de la carte de régions pour conclure. □

Cette preuve est formalisée dans Coq[11].

Lemme 5 (Analyse monotone d'une affectation)**.** *Considérons une mémoire m et une carte de régions initiale G, où la mémoire et la carte de régions sont cohérentes ($G \vdash m$). Si l'analyse d'une affectation à partir de cette carte G génère une carte de régions finale G', alors cette nouvelle carte de régions G' reste cohérente avec la mémoire m.*

Formellement :

$$G \vdash m \to \mathsf{analyse\_stmt}\ (l = e)\ G = \mathsf{Return}\ ()\ G' \to G' \vdash m$$

Démonstration. La structure de la preuve repose sur les propriétés de monotonie associées aux relations utilisées pour définir la cohérence. Il est ensuite nécessaire de démontrer cette propriété pour les fonctions impliquées dans analyse_stmt $(l = e)$, c'est-à-dire les fonctions d'analyse de l'expression, de la l-value, ainsi que celles de création de pointeur et de fusion. En appliquant ces différents lemmes de monotonie à ces fonctions, nous pouvons alors conclure. □

Cette relation n'est pas encore formalisée dans le développement Coq.

Lemme 6 (Cohérence de l'analyse d'affectation)**.** *Ce théorème stipule qu'une analyse d'affectation correcte garantit la cohérence de la carte de régions résultante avec la mémoire mise à jour. Autrement dit, si l'évaluation sémantique d'une instruction d'affectation $l = e$ appliquée à une mémoire m produit une nouvelle mémoire m', alors la carte de régions finale G', obtenue par l'analyse de l'affectation à partir de la carte de régions initiale G, est assurée de rester cohérente avec cette mémoire m'.*

Formellement, cela s'exprime comme suit :

$$G \vdash m \to [\![l = e]\!]_m = \lfloor m' \rfloor \to \mathsf{analyse\_stmt}\ (l = e)\ G = \mathsf{Return}\ ()\ G' \to G' \vdash m'$$

Démonstration. Pour prouver que la carte de régions G' est cohérente avec la mémoire m', il est nécessaire de démontrer que $G' \vdash a_2 \in r_2$ sous l'hypothèse que load $m'\ a_1\ \tau = \lfloor a_2 \rfloor$, $G' \vdash r_1 \hookrightarrow r_2$ et $G' \vdash a_1 \in r_1$.

Par analyse de cas, nous savons que : si l'expression est primitive, alors la carte de régions localise toujours les mêmes adresses, et nous pouvons conclure par le lemme 5 précédent.

Sinon, si l'expression est un pointeur, la sémantique indique que la l-value l renvoie une adresse a, et que le chargement de cette adresse nous donne une autre adresse a'.

Nous pouvons alors effectuer une analyse de cas pour déterminer si l'adresse a est séparée de l'adresse a_1. Dans le cas où elle est séparée, nous concluons par la monotonie de l'analyse. Si, en revanche, les deux adresses sont égales, nous savons que l'analyse a créé une région r dans laquelle se trouve l'adresse a, qui pointe vers une région r' où se localise a'. Nous pouvons donc conclure grâce à la relation de cohérence. □

Cette preuve n'a pas encore été formalisée dans le développement en Coq.

5.5 Correction de la sémantique

Dans cette section, nous démontrons la correction sémantique de l'analyse de région en nous appuyant sur les relations construites à partir des corrections statiques et dynamiques des sections précédentes. Ces preuves ne sont pas entièrement formalisées en Coq, mais leur développement dans l'assistant de preuve a fait naturellement apparaître des hypothèses concernant les indices de tableaux et les décalages de pointeurs. En effet, il se trouve que ces indices doivent être suffisamment *gardés* pour éviter qu'ils ne débordent des empreintes mémoires des régions associées. Nous reviendrons sur ces gardes à la section 5.6.

11. Grâce au lemme `coherence` présent dans le fichier `theories/RegionAnalysis/Representation.v`

Théorème 2 (Représentation des expressions)**.** *Pour toute mémoire m cohérente dans une carte de régions G et pour toute expression e bien gardée dans cette carte de régions, si l'évaluation de e se réduit à une valeur v, et si cette expression se localise dans une valeur abstraite d, alors la valeur v est correctement représentée dans le domaine de valeur d. Un théorème similaire existe pour les l-values et est mutuellement récursif avec ce théorème.*

$$G \vdash m \to [\![e]\!]_m = \lfloor v \rfloor \to G \vdash e : d \to G \vdash v \in d$$

Démonstration. La preuve de cette propriété repose sur une induction mutuelle sur les expressions et les l-values. Les différents cas sont traités comme suit :

- **Expressions primitives** : La preuve est triviale car la valeur se réduit directement à une valeur primitive, et le typage est immédiate.
- **Cast hétérogène de pointeurs** : On conclut directement par hypothèse d'induction sur l'expression.
- **Création d'une adresse** : On conclut par hypothèse d'induction sur les l-values, garantissant que la nouvelle adresse est bien localisée.
- **Chargement à partir d'une adresse (l-value pointeur)** : On utilise la propriété de cohérence de la mémoire pour conclure que l'adresse pointée est valide et bien représentée.

Pour les l-values, les cas sont les suivants :

- **Variable** : La preuve est immédiate, car la variable est directement assignée dans la carte de régions.
- **Accès à un champ d'une structure** : Par hypothèse d'induction, nous disposons de l'adresse de base, et l'offset du champ est déterminé par la relation d'agrégation (définition 3), faisant partie de la relation de localisation (définition 8). Nous concluons grâce à la règle *Addr-Compound* de la définition 9 sur la cohérence des valeurs.
- **Accès à un champ d'une union** : Lorsqu'un champ d'une union est accédé, la région associée à ce champ est identique à celle de l'union elle-même. Nous utilisons l'hypothèse d'induction et montrons que l'adresse du champ est équivalente à celle de la l-value de type union.
- **Accès à un tableau** : De manière similaire au cas d'accès à un champ de structure, nous utilisons la règle *Addr-Compound*. Il s'agit ici de prouver que pour tout décalage correspondant à la taille d'un élément du tableau, les adresses demeurent bien dans la région. De fait, il est impossible de prouver ce cas si on a pas de contrainte sur les indices de tableaux. Nous supposons donc que ces indices sont *gardés* et qu'ils restent dans les bornes du tableau. Nous reviendrons sur cette hypothèse dans la section 5.6.
- **Déréférencement** : Ressemble au cas des accès tableaux, à ceci près qu'on ne connaît pas statiquement le tableau auquel on accède. Ici encore, il est nécessaire que le décalage d'indice opéré dans une expression de la forme `*(p+k)` ne fasse pas sortir de pointeur `p` de son tableau d'origine. Nous supposons donc également que les indices sont bien gardés comme indiqué section 5.6.

□

Cette preuve est partiellement formalisée en Coq [12] et s'appuie sur les hypothèses d'indices bien gardés (Section 5.6).

Théorème 3 (Correction)**.** *Soit m_0 une mémoire cohérente avec la carte de régions G ($G \vdash m_0$), et S un ensemble d'affectations ($s_i : l_i = e_i$) localisées dans la carte de régions G. Pour toute exécution d'un programme contenant ces affectations : $m_0 \xrightarrow{s_1} m_1 \xrightarrow{s_2} \cdots \xrightarrow{s_n} m_n$, alors toutes les mémoires m_i sont cohérentes avec la carte de régions ($G \vdash m_i$).*

Démonstration. La preuve de ce théorème se fait par récurrence sur i. L'initialisation est donnée par l'hypothèse de cohérence de la mémoire initiale. L'étape de récurrence utilise le

12. Dans le fichier `theories/RegionAnalysis/Representation.v`

lemme 6 et l'hypothèse de récurrence sur la cohérence de la mémoire précédente. Enfin il suffit de montrer que la carte n'a pas changé car elle contient déjà les informations de cette affectation. □

On peut aussi remarquer que l'analyse de différentes instructions du programme C peut être effectuée dans n'importe quel ordre, car elle conserve les informations des analyses précédentes et ne nécessite aucun calcul de point fixe, contrairement à une analyse par interpretation abstraite par exemple.

5.6 Rôle et nécessité des gardes

Lors de la preuve en Coq du théorème 2 relatif à la représentation des valeurs sémantiques dans la carte de régions, il est apparu qu'il fallait nécessairement contraindre les indices de tableaux et les décalages de pointeurs. Prenons l'exemple de la Figure 10. Par défaut, la carte de régions générée va partionner la mémoire associée à la structure `s` en deux sous-régions distinctes, une pour le tableau `s.f[0..3]` et l'autre pour le champs `s.g`.

```
struct S { int f[4], g; };
void h(struct S s, int k, int i) {
  //@ assert 0 ≤ k < 4;
  int *p = &s.f[k];
  //@ assert 0 ≤ k + i < 4;
  *(p + i) = s.g;
}
```

Figure 10. Exemple de gardes.

Deux problèmes peuvent se produire dans cet exemple :

— Le pointeur `p = &s.f[k]` peut être localisé ou bien dans la région de `&s.f[0..3]` ou bien déborder dans le champs `g` voire à l'extérieur de la structure. Pour assurer la cohérence de la carte, il faut donc contraindre l'index `k` à rester dans l'intervalle `0..3`.

— Ensuite, on accède au pointeur `*(p+i)` ; ce décalage supplémentaire doit également être contraint pour rester dans la sous-région `s.f[0..3]`. Ce cas est plus délicat car il faut, d'une certaine manière, se souvenir du décalage initial de `p` dans cette sous-région. Au final, il faudra pouvoir contraindre l'expression `k+i` à rester dans l'intervalle `0..3`.

La Figure 10 illustre comment on pourrait générer (automatiquement) des contraintes suffisantes en ACSL pour garantir la cohérence de la carte de régions.

On peut remarquer que ces gardes ne sont pas équivalentes aux contraintes de validité des pointeurs ou de comportement bien définis, du moins, pas dans tous les modèles sémantiques du C. Ainsi, pour certains compilateurs embarqués réels, il possible d'utiliser `k+i=4` dans l'exemple ci-dessus pour accéder de manière valide au champs `s.g` via le pointeur `p`. Dans ce cas, on devra ajouter des annotations pour que l'analyse de région fusionne les régions des champs `s.f` et `s.g`, permettant ainsi de relâcher les contraintes sur les indices `i` et `k`.

La définition formelle et l'implémentation de ces gardes feront l'objet de travaux futurs. Il est cependant remarquable que ces gardes sont implicitement nécessaires pour toute analyse d'alias et que, même si elles sont parfois évoquées dans les articles, elles sont souvent confondues avec les règles de validité ou de strict-aliasing. À notre connaissance, il n'existe pas dans la littérature de définition précise de ces hypothèses ni de la manière de les vérifier.

5.7 Formalisation Coq

Pour simplifier les preuves de correction des fonctions utilisant la monade définie en section 3.4, nous introduisons des triplets de Hoare. Un triplet de Hoare est défini par la pré-condition P sur l'état initial s, l'expression monadique e produisant l'état final s' et le

résultat v, ainsi que la post-condition Q sur s' et v.

$$\begin{array}{lcl} \mathit{StateProp} & \triangleq & \forall\ \sigma,\quad \sigma \to \mathit{Prop} \\ \mathsf{triple} & : & \forall\ \sigma\ \alpha,\quad \mathit{StateProp}\ \sigma \to S\ \sigma\ \alpha \to (\alpha \to \mathit{StateProp}\ \sigma) \to \mathit{Prop} \\ \mathsf{triple} & \triangleq & \lambda\ P\ e\ Q.\quad \forall\ s\ s'\ v,\quad P\ s \to e\ s = \mathsf{Return}\ s'\ v \to Q\ v\ s' \end{array}$$

Pour simplifier son utilisation en Coq, nous introduisons la notation suivante :

```
Notation "{{  P }} e {{ r ⇒ Q }} " := (triple P e (fun r ⇒ Q)).
```

Comme nous l'avons vu la preuve de correction repose sur plusieurs propriétés de monotonie (lemme 2 et 5), qui correspondent à des invariants du programme. Pour faciliter la spécification et la vérification de ces invariants, nous définissons une fonction `invariant` qui se défini comme suit :

```
Definition invariant {σ α} (I : StateProp σ) (e : S σ α) := {{ I }} e {{ _ ⇒ I }} .
```

Les pré-conditions et post-conditions dans notre cadre sont des prédicats appartenant à *StateProp*, qui prend l'état et renvoie une proposition. Afin de simplifier l'écriture et la vérification de ces prédicats, nous introduisons plusieurs opérateurs logiques directement sur les prédicats de *StateProp*. Nous commençons par redéfinir les opérateurs classiques de la logique d'ordre supérieur afin de manipuler ces prédicats de manière concise, ansi qu'un opérateur permettant d'intégrer un prédicat pur n'utilisant pas l'état. Voici quelques exemples d'opérateurs :

```
Notation "Not P" := (fun s ⇒ ¬ P s).
Notation "Forall x .. y, P" := (fun s ⇒ ∀ x, .. (∀ y, P s) ..).
Notation "≪  P ≫ " := (fun _ ⇒ P).
```

Pour améliorer la lisibilité du contexte de preuve, nous introduisons la notation suivante, qui clarifie l'état initial, l'état final et la valeur de retour d'une opération monadique.

```
Notation "[si] e [r, sf]" := (e si = Return sf r).
```

Pour faciliter l'utilisation des triplets de Hoare dans les preuves, plusieurs tactiques ont été introduites dans Coq, disponibles dans le développement Coq.

Cet ensemble de notations et de tactiques Coq [13] facilite l'expression et la preuve des propriétés des fonctions utilisant la monade. Cela contribue également à rendre le développement et la formalisation de notre analyse de région plus concis.

Cette formalisation, implémentée en Coq, compte un peu plus de 6000 lignes de code. Une archive du projet est accessible sur *Zenodo* [LIS24] (`https://doi.org/10.5281/zenodo.14500612`).

6 Conclusion

Nous avons présenté l'implémentation et la formalisation Coq d'une analyse d'alias enrichie et modulaire permettant de cartographier précisément les accès mémoires effectués par un programme C. Les cartes de régions ainsi générées peuvent être vues comme un système de type qui permettra de définir un modèle mémoire très performant pour la preuve déductive au sein de Frama-C/WP.

C'est un travail en cours de développement, mais la formalisation en Coq a d'ores et déjà permis la mise en lumière de conditions nécessaires mais qu'il reste à définir formellement lors des travaux futurs. Notons que ces conditions sont inhérentes à toute analyse d'alias et qu'il est surprenant qu'on n'en trouve que très peu de traces dans la littérature.

Remerciements. Ce travail a bénéficié du soutien financier de l'Agence Nationale de la Recherche (projet ANR-22-CE25-0018) dans le cadre du projet CoMeMoV (`https://comemov.github.io/`).

13. Disponible dans le fichier `theories/Utils/Triplet.v`

Références

[BBB+21] Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles et Nicky Williams : The dogged pursuit of bug-free C programs : the Frama-C software analysis platform. Commun. ACM, 64(8):56–68, 2021.

[BBBC24] Allan Blanchard, François Bobot, Patrick Baudin et Loïc Correnson : Formally Verifying that a Program Does What It Should : The Wp Plug-in, pages 187–261. Springer, 2024.

[BBC+] Patrick Baudin, François Bobot, Loïc Correnson, Zaynah Dargaye et Allan Blanchard : WP Plug-in Manual.

[BDES11] Ahmed Bouajjani, Cezara Drăgoi, Constantin Enea et Mihaela Sighireanu : On inter-procedural analysis of programs with lists and data. In Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11, pages 578–589, New York, NY, USA, juin 2011. Association for Computing Machinery.

[BdMS05] Clark Barrett, Leonardo de Moura et Aaron Stump : SMT-COMP : Satisfiability modulo theories competition. In Kousha Etessami et Sriram K. Rajamani, éditeurs : Computer Aided Verification, pages 20–23, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[BFM+] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy et Virgile Prevosto : ACSL : ANSI/ISO C Specification Language.

[BMP24] Allan Blanchard, Claude Marché et Virgile Prevosto : Formally Expressing What a Program Should Do : The ACSL Language, pages 3–80. Springer, 2024.

[Bor00] Richard Bornat : Proving pointer programs in Hoare Logic. In Roland Backhouse et José Nuno Oliveira, éditeurs : Mathematics of Program Construction, pages 102–126, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[CC77] Patrick Cousot et Radhia Cousot : 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, POPL '77, page 238–252, New York, NY, USA, 1977. Association for Computing Machinery.

[Cor22] Loic Correnson : Ivette : A modern GUI for Frama-C. Lecture Notes in Computer Science, 13765:116–131, 2022. Electronic ISBN : 978-3-031-26236-4.

[CP19] Arthur Charguéraud et François Pottier : Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits. Journal of Automated Reasoning, 62(3):331–365, 2019.

[CR08] Bor-Yuh Evan Chang et Xavier Rival : Relational inductive shape analysis. In Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '08, pages 247–260, New York, NY, USA, janvier 2008. Association for Computing Machinery.

[DMM98] Amer Diwan, Kathryn S. McKinley et J. Eliot B. Moss : Type-based alias analysis. In Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation, PLDI '98, page 106–117, New York, NY, USA, 1998. Association for Computing Machinery.

[Hoa69] C. A. R. Hoare : An axiomatic basis for computer programming. Commun. ACM, 12(10):576–580, octobre 1969.

[Hor97] Susan Horwitz : Precise flow-insensitive may-alias analysis is np-hard. ACM Trans. Program. Lang. Syst., 19(1):1–6, janvier 1997.

[HT01] Nevin Heintze et Olivier Tardieu : Demand-driven pointer analysis. SIGPLAN Not., 36(5):24–34, mai 2001.

[ILR21] Hugo Illous, Matthieu Lemerre et Xavier Rival : A relational shape abstract domain. Formal Methods in System Design, 57(3):343–400, septembre 2021.

[JLRS10a] Bertrand Jeannet, Alexey Loginov, Thomas Reps et Mooly Sagiv : A relational approach to interprocedural shape analysis. ACM Trans. Program. Lang. Syst., 32(2), février 2010.

[JLRS10b] Bertrand Jeannet, Alexey Loginov, Thomas Reps et Mooly Sagiv : A relational approach to interprocedural shape analysis. ACM Trans. Program. Lang. Syst., 32(2):5 :1–5 :52, février 2010.

[KPS24] Nikolai Kosmatov, Virgile Prevosto et Julien Signoles : Guide to Software Verification with Frama-C. Springer, 2024.

[Kre12] Robbert Krebbers : The C standard formalized in Coq. Thèse de doctorat, Radboud University Nijmegen, 2012.

[LABS12] Xavier Leroy, Andrew W. Appel, Sandrine Blazy et Gordon Stewart : The CompCert Memory Model, Version 2. Research Report RR-7987, INRIA, juin 2012.

[LGRF+24] Tristan Le Gall, Jan Rochel, Florian Faissole, Julien Signoles et Denis Cousineau : Alias : pointeurs espionnés en série. In 35es Journées Francophones des Langages Applicatifs (JFLA 2024), pages 85–104, Saint-Jacut-de-la-Mer, France, janvier 2024.

[LIS24] CEA LIST : Formalisation d'une analyse de région pour Frama-C/WP, décembre 2024.

[Min06] Antoine Miné : Field-sensitive value analysis of embedded c programs with union types and pointer arithmetics. SIGPLAN Not., 41(7):54–63, juin 2006.

[SB15] Yannis Smaragdakis et George Balatsouras : Pointer Analysis. Foundations and Trends® in Programming Languages, 2(1):1–69, 2015.

[SBGG13] Neil Sculthorpe, Jan Bracker, George Giorgidze et Andy Gill : The constrained-monad problem. In Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming, ICFP '13, page 287–298, New York, NY, USA, 2013. Association for Computing Machinery.

[Ste96] Bjarne Steensgaard : Points-to analysis in almost linear time. In Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '96, pages 32–41, New York, NY, USA, janvier 1996. Association for Computing Machinery.

[SvdW10] Bas Spitters et Eelis van der Weegen : Developing the algebraic hierarchy with type classes in coq. In Matt Kaufmann et Lawrence C. Paulson, éditeurs : Interactive Theorem Proving, pages 490–493, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.