# Perle de preuve: les tableaux creux

**Romain Bardou*, Claude Marché****

* *Lab. de Recherche en Informatique, Univ Paris-Sud, CNRS, Orsay, F-91405*
** *INRIA Saclay - Île-de-France, 4 rue Jacques Monod, Orsay, F-91893*

**Résumé.** *La preuve formelle de propriétés du comportement fonctionnel des programmes impératifs est difficile car le partage de références peut invalider de manière inattendue des invariants de données. L'approche* Capucine *relève ce défi à l'aide d'un système de typage statique, basé sur les notions de régions mémoires et de permissions d'accès, qui permet de se ramener à des programmes où les effets de bords sont bien contrôlés. L'objectif de cet article est d'illustrer l'approche Capucine sur une étude de cas issue de* benchmarks *internationaux: les tableaux creux. Nous montrons comment le programme doit être annoté pour exprimer les propriétés attendues, ainsi que pour spécifier les informations de régions et de permissions. La démarche de preuve est implémentée dans un prototype, et la preuve proprement dite s'effectue avec des outils de démonstration purement automatiques, à l'exception d'un lemme que l'on prouve à l'aide de l'assistant de preuve Coq.*

**Mots-Clés :** *preuve de programme, invariants de données, partage, régions mémoire, permissions/capacités, preuve automatique, assistant de preuve Coq, benchmarks VACID-0*

## 1. Introduction

La spécification formelle permet d'exprimer des propriétés complexes sur les comportements attendus des programmes. Quand les spécifications sont exprimées dans un langage logique expressif, disons au minimum la logique du premier ordre, il faut faire appel à des techniques de preuve pour démontrer qu'un programme satisfait ses spécifications.

Dans le cas de programmes purement applicatifs, les techniques de preuve sont relativement bien maîtrisées, car le langage d'écriture des

programmes est proche de la logique. Des logiques très expressives comme le calcul des constructions inductives, implémenté dans le système Coq [BC04], permettent ainsi d'exprimer programmes et spécifications dans le même langage, mais ces programmes doivent être purement applicatifs.

La logique de Floyd-Hoare [Flo67, Hoa69] et le calcul de plus faible précondition de Dijkstra [Dij76] sont des approches bien connues pour prouver des propriétés de programmes contenant des effets de bords. Néanmoins, ces approches font une hypothèse implicite souvent mal identifiée, qui est que les références (ou variables modifiables en place) ne peuvent être *aliasées*, autrement dit que le partage de références est interdit. Cette hypothèse doit être bien comprise en particulier dans le cas de la preuve *modulaire* de programmes formés de plusieurs sous-programmes. Ceci peut être illustré par le petit exemple suivant (en OCaml pour fixer les idées) muni d'une post-condition.

```
let f (x:int ref) (y:int ref) =
  x := !x + 1;
  y := !y + 1
{ !x = old(!x) + 1 and !y = old(!y) + 1 }
```

La validité de cette post-condition est établie sous l'hypothèse implicite que $x$ et $y$ sont distinctes. Ainsi un appel à f de la forme

```
let z = ref 0 in f z z
```

doit être interdit (sinon la post-condition de f dirait que z est incrémenté de 1, au lieu de 2). L'outil Why [FM07] traite un tel cas grâce à un système de types qui rejette l'appel à `f` ci-dessus. Sous cette hypothèse rendue explicite, Filliâtre [Fil03] a montré la correction d'une logique de Hoare, en se ramenant à des programmes purs.

La preuve de programmes permettant les alias est donc restée longtemps moins étudiée et donc moins comprise que pour les programmes sans alias. Or les besoins dans ce domaine se sont manifestés, avec l'apparition d'outils pour traiter les programmes de type Java ou C : ESC/java [CK04], Spec# [BLS04], KeY [BHS07], VCC [DMS$^+$09],

Why [FM07], Frama-C [Fra08], etc. Dans ce contexte, la génération d'obligations de preuves est effectuée en se ramenant au cas sans alias, en construisant des modèles de la mémoire : par exemple, le tas mémoire peut être vu comme un grand tableau indexé par les pointeurs, ou, plus subtilement, le modèle *component-as-array* [Bor00] voit chaque champ de structure comme un grand tableau. La difficulté principale qui apparaît est alors le besoin de spécifier en permanence les alias de pointeurs potentiels. Les défis à résoudre ont été bien définis par Leavens, Leino et Müller en 2007 [LLM07]. Un des défis est de pouvoir raisonner sur des structures de données complexes, modifiables en place, avec des invariants de données à préserver.

En 2010, une collection de programmes a été proposée par Leino et Moskal [LM10] : les *benchmarks VACID (Verification of Ample Correctness of Invariants of Data-structures)*, disponibles sur la page Web `http://vacid.codeplex.com/`. Ces exemples sont des programmes courts qui illustrent les défis pour les approches de preuve modulaire de programmes prétendant supporter les données avec partage.

L'objectif de cet article est d'illustrer une nouvelle approche que nous proposons pour traiter les programmes avec pointeurs et en particulier le partage. Cette approche appelée *Capucine*, implémentée dans un outil du même nom, est détaillée dans un rapport de recherche [BM10]. Nous allons décrire informellement cette approche sur le premier exemple de la collection VACID : les tableaux creux.

Dans la section 2, nous présentons ce cas d'étude en détail, ainsi que les défis qu'il pose. La section 3 expose brièvement les principes de l'approche Capucine, sur le cas simple des tableaux standards. La section 4 montre ensuite, en plusieurs étapes successives, comment le programme d'étude et ses spécifications sont modélisés, et comment la preuve est effectuée. La section 5 conclut en comparant avec des travaux proches et donne des perspectives.

```
class SparseArray {
  static final int DEFAULT = 0;
  int val[];
  uint idx[], back[];
  uint n, uint size;

  static SparseArray create(uint sz) {
    SparseArray t = new SparseArray();
    val = new int[sz];
    idx = new uint[sz];
    back = new uint[sz];
    n = 0;
    size = sz;
    return t;
  }

  int get(uint i) {
    if (idx[i] < n && back[idx[i]] == i) return val[i];
    else return DEFAULT;
  }

  void set(uint i, int v) {
    val[i] = v;
    if (!(idx[i] < n && back[idx[i]] == i)) {
      assert(n < size);                    /* assertion (1) */
      idx[i] = n; back[n] = i; n = n + 1;
    }
  }

  static void sparseArrayTestHarness() {
    SparseArray a = create(10), b = create(20);
    assert(a.get(5) == DEFAULT && b.get(7) == DEFAULT);
    a.set(5, 1); b.set(7, 2);
    assert(a.get(0) == DEFAULT && b.get(0) == DEFAULT);
    assert(a.get(5) == 1 && b.get(7) == 2);
    assert(a.get(7) == DEFAULT && b.get(5) == DEFAULT);
  }
}
```

Figure 1 – Code du challenge SparseArray.

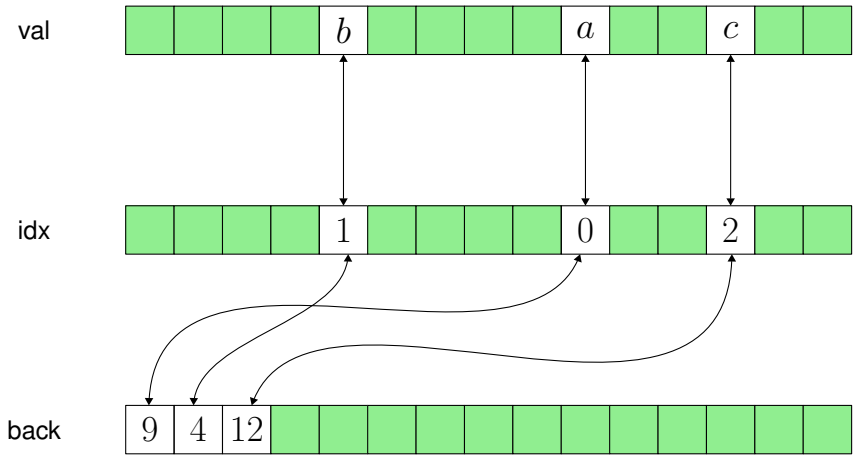


Figure 2 – Vision schématique d'un tableau creux, pour $size = 15$ et $n = 3$.

## 2. Les tableaux creux

Le défi *constant-time sparse arrays* des benchmarks VACID est posé sur le code de la figure 1. Celui-ci est écrit dans une syntaxe proche de celle de Java, mais la présence de classes n'est pas importante et, comme indiqué dans l'article décrivant ces benchmarks [LM10], on suppose que les tableaux alloués à l'aide de `new` ne sont pas initialisés[1].

La classe `SparseArray` implémente une structure de *tableaux creux*. Elle fournit les fonctions usuelles `create`, `get` et `set`. La particularité de ces tableaux creux est que tout comme la lecture `get` et l'affectation `set`, l'allocation `create` a lieu en temps constant (indépendemment de la taille). En effet, `create` n'initialise pas les tableaux internes, mais néanmoins l'accès `get` à un indice non initialisé renvoie la valeur `DEFAULT`.

Les tableaux creux sont implémentés à l'aide de trois tableaux `val`, `idx` et `back`, et de deux entiers `size` et `n`. Le tableau `val` contient les valeurs effectives du tableau creux. La valeur `n` est le nombre de cases

1. Nous avons légèrement modifié le code original, en remplaçant l'allocation implicite des tableaux internes, spécifique à Java, par des allocations explicites dans le constructeur. Cela permet par ailleurs de créer des tableaux d'une taille `sz` donnée en paramètre, plutôt que d'une taille constante `MAXLEN`.

différentes qui ont été affectées par `set`. Le tableau `back` contient, dans les cases $0$ à $n-1$, les indices de ces cases affectées. Le tableau `idx` permet de savoir où dans `back` ces indices apparaissent. Ainsi, les cases $0$ à $n-1$ de `back` sont en bijection avec les cases correspondantes de `idx`.

La figure 2 illustre l'état de la structure après l'ajout de trois éléments $a$, $b$ et $c$, dans cet ordre, aux indices $9$, $4$ et $12$ respectivement. Les cases plus foncées sont celles qui n'ont pas encore été affectées. Le premier élément $a$, d'indice 9, à été associé à l'indice 0 dans le tableau $idx$, et son véritable indice $9$ est stocké à l'indice $0$ du tableau $back$.

Cette structure de données permet d'accéder à un élément inséré précédemment à un indice $d$ de la façon suivante (code de la méthode `get`) : (a) lire l'indice interne $i$ stocké dans $idx[d]$, (b) si on n'a pas $0 \leq i < n$ alors à coup sûr aucun élément d'indice $d$ n'a jamais été inséré, donc il faut renvoyer la valeur par défaut, (c) si $0 \leq i < n$ alors on regarde à l'indice $d' = back[i]$, (d) si $d' = d$ alors il y a effectivement un élément d'indice $d$ inséré, donc on retourne $val[d]$, sinon on retourne la valeur par défaut.

Les vérifications à effectuer sont données par les assertions du code de la figure 1. Une première assertion apparaît dans la méthode `set()`, étiquetée par (1). Les autres assertions apparaissent dans la méthode `sparseArrayTestHarness()` qui permet de tester l'implémentation sur des données simples. Cela peut sembler facile au premier abord — en effet il suffirait d'exécuter le programme pour tester la validité des assertions — mais le but ici est d'effectuer une vérification statique et modulaire. Il faut donc être capable de spécifier un contrat (pré- et post-condition) sur les méthodes `get()` et `set()`, suffisant pour établir les assertions.

Remarquons que nous avons ajouté une assertion supplémentaire par rapport au code VACID d'origine [LM10] : la dernière, la seule qui teste que $a$ et $b$ sont bien séparés. En effet, ce qui rend la tâche plus ardue encore est que la méthode `create()` doit être spécifiée d'une manière qui garantisse que $a$ et $b$ créés dans `sparseArrayTestHarness()` ne partagent pas de données. Par exemple, comment assurer que $a.idx$ et $b.idx$ ne sont pas identiques ? Cela signifie que nous avons besoin d'une

méthodologie qui apporte des informations de séparation. C'est ce que propose notre approche Capucine, avec un typage à base de régions et de permissions.

## 3. L'approche Capucine en bref

Le langage de Capucine est un langage dédié à la preuve de programme, volontairement réduit aux constructions essentielles pour l'approche.

### 3.1. *Types et fonctions logiques*

Seuls les types des booléens, des entiers (mathématiques, c.-à-d. non bornés) et des n-uplets sont prédéfinis. L'utilisateur peut en revanche introduire de nouveaux types purs grâce à des axiomatisations en logique du premier ordre (c'est le même principe que celui de l'outil Why).

Par exemple, il n'y a aucune notion de tableau en Capucine, mais on peut les définir en commençant par axiomatiser les tableaux purs infinis de la façon suivante (c'est la *théorie des tableaux* classique de la déduction automatique) :

```
logic type array (alpha)
logic function
  store (array (alpha), int, alpha): array (alpha)
logic function select (array (alpha), int): alpha

axiom select_eq: forall a: array (alpha). forall i: int.
  forall v: alpha. [select(store(a, i, v), i)] = [v]

axiom select_neq: forall a: array (alpha). forall i: int.
  forall j: int. forall v: alpha.
  [i] <> [j] ==>
    [select(store(a, i, v), j)] = [select(a, j)]
```

Notons que l'on déclare ici des tableaux polymorphes (argument `alpha` du type `array`). La notation entre crochets est la syntaxe Capucine permettant de différencier les termes des assertions dans la logique.

### 3.2. *Classes*

Les données modifiables en place, que l'on va manipuler dans les programmes, sont déclarées comme des références sur un type précédemment déclaré, et accompagnées d'un invariant de données. On appelle *classe* cette combinaison d'une référence et d'un invariant[2].

Ainsi, les tableaux non purs, c.-à-d. modifiables en place, sont définis comme une référence sur une paire formée d'un entier — la longueur, positive ou nulle — et d'un tableau pur :

```
selector (length, cell)
class Array (alpha) =
  (int * array (alpha))
  invariant(this) = [0] <= [this.length]
end
```

L'information de longueur va nous permettre de vérifier que les accès sont dans les bornes. Remarquons qu'en Capucine, l'accès aux composantes d'un $n$-uplet s'écrit $e.1, \ldots, e.n$. La déclaration `selector` ci-dessus introduit un sucre syntaxique pour accéder à ces composantes en donnant des noms plus explicites : sur l'exemple `this.length` est équivalent à `this.1`.

### 3.3. *Définitions de fonctions, régions mémoires et typage*

La définition d'une fonction Capucine se fait dans une syntaxe proche de celle d'OCaml. On accède aux références avec `!` et on les met à jour avec `:=`. Par ailleurs, on peut attacher une précondition et une postcondition à ces fonctions.

Continuons notre exemple en donnant la fonction d'accès à une case d'un tableau :

```
val array_get(a: Array (alpha) [R], i:int) : alpha
```

2. Cette notion ne doit pas être confondue avec la notion de classe dans les langages orientés objets : il n'y pas de notion d'héritage ou d'appel dynamique en Capucine. On l'utilise seulement par analogie avec JML ou Spec#, où les invariants sont attachés aux classes.

```
  requires [0] <= [i] and [i] < [!a.length]
  ensures [result] = [select(!a.cell,i)]
  =
    select(!a.cell,i)
```

La précondition (mot-clé `requires`) rejette les accès en dehors des bornes, et la postcondition (`ensures`) indique simplement que le résultat est le contenu de la bonne cellule.

On voit apparaître ci-dessus la notion de *région mémoire*. En Capucine, toute référence doit appartenir à une certaine région, qui est indiquée dans son type. Ici, le type du premier argument de `array_get` est `Array(alpha)[R]` : un pointeur, de type `Array(alpha)`, dans la région `R`. De même que `alpha` est une variable de type, `R` est ici une variable de région, implicitement universellement quantifiée.

Le type d'une expression dans un programme est toujours soit un type logique ; soit une référence, c.-à-d. une classe annotée par une région ; soit un $n$-uplet de type de programmes. Pour que deux types références soient égaux, il faut non seulement qu'ils aient la même classe mais aussi la même région.

### 3.4. *Permissions et mise à jour d'une référence*

L'intérêt des régions se manifeste avec les permissions d'accès : en Capucine, la mise à jour d'une référence n'est autorisée que si la permission sur la région de cette référence est disponible. Les permissions disponibles sont indiquées dans le code.

Par exemple, la mise-à-jour d'un tableau se code en Capucine par :

```
val array_set(a: Array (alpha) [R], i:int, v:alpha) : unit
  consumes R^c
  produces R^c
  requires [0] <= [i] and [i] < [!a.length]
  ensures [!a.length] = [old(!a.length)] and
          [!a.cell] = [store(old(!a.cell),i,v)]
  =
    unpackregion R;
```

```
a := (!a.length, store(!a.cell,i,v));
packregion R
```

Le code consiste à changer la paire pointée par la référence à l'aide de l'opération de mise à jour fonctionnelle, ce qui est reflété dans la post-condition. La construction `consumes` indique les permissions qui doivent être fournies à la fonction, et `produces` indique celles qui sont rendues à l'appelant. Cette notion de permission est statique : elle fait partie des règles de typage de Capucine. Le point important est que les permissions ne peuvent pas être dupliquées : ce typage est *linéaire*. Autrement dit, pour que l'appelant puisse disposer encore de la permission qu'il a donnée à l'appel, il faut la lui rendre.

Sur l'exemple ci-dessus, la permission consommée et produite `R^c` dénote plus précisément que `R` est une *région singleton fermée*. Cela veut dire que la région `R` ne contient qu'un seul pointeur, et « fermé » précise que son invariant est valide. La mise à jour de la valeur pointée par une référence requiert la permission *ouverte*. Comme la permission fournie est fermée, on utilise la construction spéciale `unpackregion` : elle est sans effet si ce n'est de consommer `R^c` et de produire `R^o`, la permission sur la région `R` singleton ouverte. Le typage autorise alors l'affectation. Enfin, pour pouvoir produire de nouveau la permission fermée, on doit refermer la région avec `packregion`.

Le point important ici est que la fermeture de la région n'est pas anodine : l'invariant de données *doit être validé à ce moment-là*. En pratique, on va voir que cela générera une obligation de preuve.

### 3.5. *Régions vides*

Le langage des permissions contient la syntaxe `R^e` pour dénoter une région vide : `R` ne contient aucun pointeur. Cette permission est attendue et consommée par l'allocation d'une référence, qui ne peut avoir lieu que dans une région vide.

Toujours sur l'exemple des tableaux, la création d'un tableau d'une longueur donnée est définie de la façon suivante :

```
val array_create(size:int): Array (alpha) [R]
```

```
consumes R^e
produces R^c
requires [0] <= [size]
ensures [!result.length] = [size]
=
  let tmp = new Array (alpha)[R] in
  tmp := (size, !tmp.cell);
  packregion R;
  tmp
```

L'opération `new` consomme `R^e` et produit `R^o`. De nouveau, l'opération `packregion` va demander de vérifier l'invariant, et produit `R^c`.

### 3.6. *Régions groupes, adoption et focus*

Les régions peuvent contenir plusieurs références. L'opération d'adoption permet d'ajouter une référence d'une région singleton fermée à une autre région fermée (dont les références sont de même type). On obtient une région groupe, dont la permission est dénotée `R^g`. Il faut noter qu'une région groupe est toujours fermée : ses références doivent donc respecter leurs invariants.

On ne peut pas mettre à jour une référence d'une région groupe `R^g` directement. Il faut commencer par faire un *focus* sur la référence voulue, ce qui consomme `R^g` et produit d'une part une région singleton fermée `S^c` et la dernière sorte de permission possible, dénoté `S -o R` qui indique que `R` est actuellement soumise à un tel focus sur une sous-région `S`.

### 3.7. *Régions possédées par une référence*

Une déclaration de classe peut contenir des déclarations de régions *possédées*. Les permissions de ces régions sont alors encapsulées dans la référence elle-même, c.-à-d. que pour en disposer il faut d'abord ouvrir la référence en question.

Par exemple, voici la déclaration d'une référence pointant sur un tableau d'entiers ainsi que la somme de ces éléments.

```
selector (sum,values)
class Calc =
  own Ra;
  (int * Array(int)[Ra])
  invariant(this) =
    [this.sum] =
    [sum(0,!(this.values).length - 1,!(this.values).cell)]
end
```

Le point important ici est que les invariants ne peuvent accéder qu'aux références des régions possédées. Cette condition est essentielle pour garantir que la méthodologie va préserver les invariants de données. Par exemple, si `p` est de type `Calc[R]` avec `R` fermée, il n'est pas autorisé de modifier directement le tableau `!p.values` ci-dessus (ce qui pourrait rompre l'invariant) : il faut d'abord ouvrir `p`, puis ouvrir `p.values`, le modifier et le refermer, puis enfin refermer `p`, en vérifiant que l'on rétablit l'invariant.

La table suivante résume les permissions consommées et produites par les opérations spéciales, où $R_{own}$ désigne les régions possédées par R.

| | consomme | produit |
|---|---|---|
| new | $R^e$ | $R^c, R^e_{own}$ |
| unpack | $R^c$ | $R^o, R^g_{own}$ |
| pack | $R^o, R^g_{own}$ | $R^c$ |
| focus | $R^g$ | $S^c, S \multimap R$ |
| unfocus | $S^c, S \multimap R$ | $R^g$ |
| adopt | $R^g_1, R^g$ | $R^g$ |

Enfin, il existe des affaiblissements implicites, par exemple d'une région singleton en une région groupe. Les détails sont disponibles dans [BM10].

### 3.8. *Sureté de l'approche Capucine*

La correction de cette approche est démontrée par un théorème de cohérence [BM10] qui énonce un invariant au niveau méta de l'exécution d'un programme Capucine : la préservation d'un certain nombre

de propriétés attendues de la sémantique opérationnelle. Par exemple, quand une permission singleton est disponible, la région correspondante contient effectivement un seul pointeur. La propriété qui nous intéresse le plus est la suivante : l'invariant de toute référence d'une région fermée est valide.

### 3.9. *Inférence*

Nous n'avons pas encore montré d'exemples d'utilisation de *adopt*, *focus* et *unfocus*. Très rapidement, ces indications sont très lourdes à indiquer, et pour la plupart sont systématiques. D'un point de vue pratique, notre implémentation intègre un mécanisme d'inférence de ces « annotations » : pour un programme donné, le typage ne se contente pas de décider si les types et permissions indiquées respectent les règles, mais tente aussi d'insérer les opérations *adopt*, *unfocus*, *pack* et *unpack* qui peuvent le rendre bien typé s'il ne l'est pas.

L'inférence est là pour soulager le programmeur, elle n'est pas complète (c.-à-d. ne permet pas forcément de trouver les bonnes opérations quand elle existent) ni principale (il n'y a pas toujours une unique meilleure façon d'ajouter de telles opérations). C'est pourquoi le programmeur peut toujours écrire ces opérations explicitement.

L'inférence n'interfère pas avec la sûreté de la méthode. En effet, le programme transformé diffère du programme original uniquement par les opérations sus-mentionnées, qui ne modifient pas la mémoire. Ils ont donc la même sémantique opérationnelle.

### 3.10. *Interprétation vers Why*

Pour générer les obligations de preuve, on génère un programme Why intermédiaire équivalent. Chaque région $\rho$ est traduite par trois références Why :

– $G_\rho$, un tableau qui associe des valeurs à des pointeurs ;

– $P_\rho$, l'unique pointeur de $\rho$ si celle-ci est singleton ;

– $V_\rho$, la valeur associée à l'unique pointeur de $\rho$ si celle-ci est singleton.

Le tableau $G$ n'est utilisé que si la région est groupe, sinon on utilise $P$ et $V$ à la place. Il s'agit d'une optimisation permettant d'éviter une indirection dans le cas des régions singletons, ce qui simplifie les obligations de preuve. Le pointeur $P$ est aussi utilisé pour garder une trace des pointeurs qui ont subit un *focus*. Lors du *focus* d'un pointeur $p$ d'une région $\rho$ vers une région $\sigma$, la région $\sigma$ était vide et devient singleton. On a donc $P_\sigma = p$. Cette valeur ne sera plus jamais changée, et donc on sait que si la permission $\sigma \multimap \rho$ est disponible, alors $P_\sigma$ est le pointeur qui résulte de ce *focus*.

Ainsi, pour traduire un déréférencement $!p$ on regarde les permissions disponibles. Si $p$ est dans $\rho$ par typage, on traduit $!p$ de la façon suivante :

– si $\rho^c$ est disponible, $\rho$ est singleton et donc $!p$ devient $!V_\rho$ ;

– si $\rho^g$ est disponible, $\rho$ est groupe et donc $!p$ devient $get(p, !G_\rho)$ où $get$ est la fonction d'accès aux tableaux ;

– si $\sigma \multimap \rho$ est disponible, $!p$ devient un saut conditionnel `if` : si $p = !P_\sigma$, alors on lit $p$ comme s'il était dans $\sigma$, sinon $!p$ devient $get(p, !G_\rho)$ ;

– si aucune permission n'est disponible, on traduit $!p$ par une *boîte noire* Why, autrement dit une valeur inconnue.

Pour prouver la correction de cette traduction, on définit une relation entre un tas mémoire Capucine et un tas mémoire Why. Ensuite on prouve, étant donné un tas Capucine $H$ et un tas Why $H_w$ en relation, que si le programme Why se réduit à partir de $H_w$, alors le programme Capucine se réduit à partir de $H$. De plus, les tas obtenus sont en relation.

Un point important que nous voulons souligner concerne les obligations de preuve d'invariant : celles-ci sont présentes comme annoncé au moment d'une opération *pack*. Mais également les invariants sont automatiquement insérés comme hypothèses en entrée de fonction (mais uniquement pour les régions dont la permission est donnée bien entendu). Elles sont également données en hypothèses juste après l'appel d'une fonction. Cela diffère de façon essentielle de l'approche plus classique qui interprète ces invariants comme des préconditions et postconditions systématiques : si l'on fait cela, il est à la charge de l'appelant de prouver ces invariants, et de même ces invariants doivent être

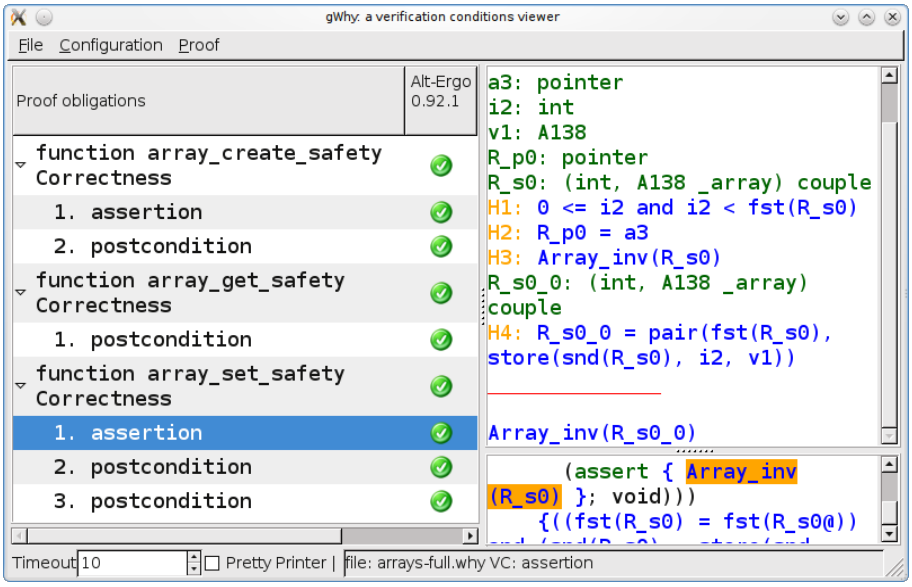


Figure 3 – Obligations de preuve pour les tableaux dans l'interface graphique de Why.

prouvés en sortie de fonction. Toutes ces obligations de preuves sont inutiles avec notre méthodologie qui garantit les invariants statiquement, à l'aide du typage des permissions.

Pour illustrer la génération d'obligations de preuve, la figure 3 montre celles générées sur nos fonctions sur les tableaux. Celles-ci sont prouvées automatiquement par l'ensemble des prouveurs automatiques Alt-Ergo, Simplify, Z3 et CVC3 qui sont à l'heure actuelle les quatre meilleurs prouveurs SMT supportant les quantificateurs. L'obligation de preuve surlignée est celle qui demande de prouver l'invariant lors du *pack* dans `array_set`.

## 4. Les tableaux creux en Capucine

### 4.1. *Structure de données et invariant*

La déclaration Capucine correspondant à la classe Sparse du code de la figure 2 est la suivante :

```
predicate interval(a:int, x:int, b:int) =
  [a] <= [x] and [x] < [b]

selector (value, idx, back, n, default, size)
class Sparse (a) =
 own Rval, Ridx, Rback;
 (Array (a) [Rval] * Array (int) [Ridx] *
  Array (int) [Rback] * int * a * int)
 invariant (x) =
  [0] <= [x.n] and [x.n] <= [x.size] and
  [!(x.value).length] = [x.size] and
  [!(x.idx).length] = [x.size] and
  [!(x.back).length] = [x.size] and
  forall i: int. interval([0],[i],[x.n]) ==>
   interval([0],[select (!(x.back).cell, i)],[x.size]) and
   [select (!(x.idx).cell,
            select (!(x.back).cell, i))] = [i]
end
```

L'invariant formalise le diagramme de la figure 1. Contrairement au code pseudo-Java, on traite ici des tableaux polymorphes, et c'est pour cela que la valeur par défaut n'est pas globale.

### 4.2. *Le modèle abstrait d'un tableau creux*

Les propriétés fonctionnelles de notre programme vont être exprimées sur un modèle abstrait d'un tableau creux, de la façon suivante.

```
predicate is_elt(a: Sparse (a) [R], i: int) =
 [0] <= [select (!(!a.idx).cell, i)] and
 [select (!(!a.idx).cell, i)] < [!a.n] and
 [select (!(!a.back).cell,
          select (!(!a.idx).cell, i))] = [i]

logic function model (Sparse (a) [R], int): a

axiom model_in:
 forall a: Sparse (a) [R]. forall i: int.
  is_elt([a], [i])
```

```
   ==> [model(a, i)] = [select(!(!a.value).cell, i)]

axiom model_out:
 forall a: Sparse (a) [R]. forall i: int.
  not is_elt([a], [i]) ==> [model(a, i)] = [!a.default]
```

Le prédicat `is_elt`$(a, i)$ indique si l'indice $i$ correspond à une valeur définie du tableau $a$. La fonction logique `model` décrit le modèle d'un tableau par la fonction des indices vers les éléments qu'il représente : plus précisément, `model`$(a, i)$ retourne l'élément associé à l'indice $i$ dans le tableau creux $a$.

### 4.3. *Fonctions* `create()`, `get()` *et* `set()`

Le code Capucine pour la fonction de création d'un tableau creux est

```
val create(sz:int, def: alpha): Sparse (alpha) [R]
  consumes R^e
  produces R^c
  requires [0] <= [sz]
  ensures  [!result.size] = [sz] and
           forall i: int. [model (result, i)] = [def]
  = let arr = new Sparse (alpha) in
    arr := (array_create (sz), array_create (sz),
            array_create (sz), 0, def, sz);
    arr
```

Elle attend une région vide et renvoie une région singleton fermée contenant le nouveau tableau creux. On n'indique pas le `packregion` qui est inféré. Le modèle résultat est la fonction constante égale à la valeur par défaut `def`.

La fonction d'accès est

```
val get(a: Sparse (alpha) [R], i: int): alpha
  consumes R^c
  produces R^c
  requires [0] <= [i] and [i] < [!a.size]
  ensures [result] = [model(a, i)]
```

```
  = let index = array_get(!a.idx, i) in
    if 0 <= index and index < !a.n and
              array_get(!a.back, index) = i
    then array_get(!a.value, i)
    else !a.default
```

Il est à noter qu'elle a besoin de la permission même si elle ne modifie rien, car elle a besoin de savoir que l'invariant est vrai.

La fonction de mise à jour est

```
val set(a: Sparse (alpha) [R], i: int, v: alpha): unit
  consumes R^c
  produces R^c
  requires [0] <= [i] and [i] < [!a.size]
  ensures
    [!a.size] = [old(!a.size)] and
    (forall j: int. [j] <> [i] ==>
       [model(a, j)] = [old(model(a, j))])
    and [model(a, i)] = [v]
  = array_set((focus !a.value), i, v);
    let index = array_get(!a.idx, i) in
    if not (0 <= index and index < !a.n and
            array_get(!a.back, index) = i)
    then (
      assert [!a.n] < [!a.size];        (* assertion (1) *)
      array_set((focus !a.idx), i, !a.n);
      array_set((focus !a.back), !a.n, i);
      a := (!a.value, !a.idx, !a.back,
            !a.n + 1, !a.default, !a.size);
    )
```

Comme on met à jour les tableaux qui composent la structure, on a besoin d'indiquer les focus, par contre les adopt et unfocus sont inférés. La post-condition indique abstraitement l'effet escompté en indiquant comment le modèle est modifié.

### 4.4. *Fonction* `testHarness()`

La fonction de test se code alors ainsi :

```
val testHarness(): unit =
  region Ra: Sparse (int) in
  region Rb: Sparse (int) in
  let default = 0 in
  let a = (create(10,default): Sparse (int) [Ra]) in
  let b = (create(20,default): Sparse (int) [Rb]) in
  let x = get(a, 5) in
  let y = get(b, 7) in
  assert ([x] = [default] and [y] = [default]);
  set(a, 5, 1);
  set(b, 7, 2);
  let x = get(a, 5) in
  let y = get(b, 7) in
  assert ([x] = [1] and [y] = [2]);
  let x = get(a, 7) in
  let y = get(b, 5) in
  assert ([x] = [default] and [y] = [default]);
  let x = get(a, 0) in
  let y = get(b, 0) in
  assert ([x] = [default] and [y] = [default]);
  let x = get(a, 9) in
  let y = get(b, 9) in
  assert ([x] = [default] and [y] = [default]);
  assert false (* test du pauvre pour la cohérence *)
```

Noter que l'on doit indiquer les régions dans lesquelles on crée les tableaux : l'inférence ne peut pas deviner que l'on veut deux régions distinctes.

### 4.5. *Preuve des obligations générées*

L'exécution de l'outil Capucine sur le fichier complet engendre 62 obligations de preuve[3], et les réponses des prouveurs automatiques sur celles-ci sont indiqués sur la figure 4. Toutes les obligations des fonc-

3. Les conjonctions sont automatiquement découpées en plusieurs obligations, ce qui explique ce grand nombre. De plus, la conjection paresseuse de la condition du `if` dans le code de `set()` multiple encore le nombre d'obligations.

| fonction | nombre d'obligations | Alt-Ergo | Simplify | Z3 | CVC3 | non prouvées |
|---|---|---|---|---|---|---|
| `create` | 4 | 4 | 4 | 4 | 4 | 0 |
| `get` | 6 | 6 | 6 | 6 | 6 | 0 |
| `set` | 25 | 22 | 23 | 23 | 22 | 2 |
| `testHarness` | 27 | 20 | 26 | 26 | 26 | 1 |

Figure 4 – Obligations de preuve pour les tableaux creux.

tions `create()` et `get()` sont prouvées, par les quatre prouveurs. Pour `set()`, la plupart des obligations sont prouvées par tous les prouveurs, sauf une prouvée seulement par Simplify et Z3 : la post-condition de `set()` parlant du modèle ; et deux occurrences de l'assertion (1), non prouvées (la duplication étant due à la conjonction paresseuse du if). L'assertion (1) est en effet difficile à prouver et nous l'étudions plus en détail dans la section 4.6 ci-après.

Pour `testHarness()`, toutes les obligations sont prouvées[4] à part le `assert false` final, mis ici pour détecter un éventuel problème de cohérence des spécifications, il est donc heureux que cela ne soit pas prouvé. Ce test de cohérence n'est bien sûr pas un critère complet, mais cela permet déjà de se rassurer : en effet, cela montre que les preuves précédentes effectuées par les prouveurs automatiques ne sont pas obtenues par une contradiction dans le contexte, par exemple due aux axiomes introduits[5]. Remarquons que pour garantir formellement la cohérence, il faudrait mettre en œuvre des techniques nettement plus lourdes, comme par exemple réaliser entièrement l'axiomatique en Coq.

4. Alt-Ergo montre quelques faiblesses, dues au moins en partie à Why, qui seront en principe corrigées dans les prochaines versions distribuées.
5. L'expérience montre que ce test est utile pour détecter des incohérences lors de la mise au point des spécifications.

```
(* array maps [0;n[ into [0;k[ *)
predicate into(n:int, k:int, t:array(int)) =
  forall i:int. interval(0,i,n) ==>
    interval(0,[select(t,i)],k)

predicate injective(n:int, t:array(int)) =
  (forall i:int. forall j:int.
    interval(0,i,n) ==> interval(0,j,n) ==>
      [select(t,i)] = [select(t,j)] ==> i = j)

predicate surjective(n:int, t:array(int)) =
  forall k:int. interval(0,k,n) ==>
    exists j:int. interval(0,j,n) and [select(t,j)] = k

lemma array_injective_is_surjective:
  forall n:int. forall a:array(int).
  into(n,n,a) and injective(n,a) ==> surjective(n,a)
```

Figure 5 – Lemme auxiliaire pour la preuve de l'assertion difficile.

### 4.6. *Preuve de l'assertion difficile*

La preuve de l'assertion étiquetée (1) sur le code de la figure 1, qui permet d'assurer que l'on n'accède pas en dehors des bornes du tableau `back`, demande un raisonnement assez subtil, inaccessible aux prouveurs automatiques. La preuve informelle est la suivante : par l'absurde, supposons que $n = size$ au point de programme où est placé le assert. Alors d'après l'invariant des tableaux creux, illustré par la figure 2, le tableau `back` est complètement rempli, et comme les tableaux `idx` et `back` sont inverses l'un de l'autre, ils représentent alors une bijection de l'ensemble $\{0..size-1\}$. Mais alors obligatoirement toutes les cases de `idx` sont remplies également, donc la valeur de $i$ au point de programme en question désigne un indice qui est soit en dehors des bornes, soit déjà occupé, ce qui contredit la condition de l'instruction if.

Pour formaliser ce raisonnement, on pose un lemme général sur les tableaux d'entiers : si un tel tableau envoie $\{0..n-1\}$ dans $\{0..n-1\}$ et que cette fonction est injective, alors elle est aussi surjective. Ceci est

```
val set(a: Sparse (a) [R], i: int, v: a): unit
  consumes R^c
  produces R^c
  requires [0] <= [i] and [i] < [!a.size]
  ensures
    [!a.size] = [old(!a.size)] and
    (forall j: int. [j] <> [i] ==>
        [model(a, j)] = [old(model(a, j))])
    and ([model(a, i)] = [v])
  =
    array_set((focus !a.value), i, v);
    let index = array_get(!a.idx, i) in
    if not (0 <= index and index < !a.n and
            array_get(!a.back, index) = i)
    then (
      (* array back maps [0 ;n[ into [0 ;size[ *)
      assert into([!a.n],[!a.size],[!(!a.back).cell]); (*A*)
      (* array back is injective *)
      assert injective([!a.n],[!(!a.back).cell]);      (*B*)
      (* if n=size then array back is also surjective *)
      assert [!a.n] = [!a.size] ==>
               surjective([!a.n],[!(!a.back).cell]);   (*C*)
      (* then i is out of bounds, or already filled *)
      assert [!a.n] = [!a.size] ==>
       interval(0,i,[!a.n]) ==>
        exists j:int.
          interval(0,j,[!a.n]) and
          [select(!(!a.back).cell,j)] = i;            (*D*)
      (* the hard assertion *)
      assert [!a.n] < [!a.size];                      (*1*)
      array_set((focus !a.idx), i, !a.n);
      array_set((focus !a.back), !a.n, i);
      a := (!a.value, !a.idx, !a.back,
            !a.n + 1, !a.default, !a.size)
    )
```

Figure 6 – Assertions auxiliaires pour la preuve de l'assertion difficile.

posé dans le code source, comme indiqué sur la figure 5. En principe, ce lemme suffit pour finir la preuve de l'assertion avec un raisonnement au premier ordre, mais en pratique les prouveurs automatiques ne sont pas assez puissants pour y parvenir, aussi une méthode standard consiste à rajouter dans le code source des assertions intermédiaires, jouant le rôle de coupures pour guider ces prouveurs. Le code annoté qui en découle est montré sur la figure 6.

Avec ces annotations supplémentaires, on obtient 33 obligations pour la fonction `set()`. Celles-ci sont prouvées automatiquement : complètement par Simplify, toutes sauf 2 pour Z3, sauf 3 pour CVC3 et sauf 7 pour Alt-Ergo. En fait, les assertions (A) et (B) ne sont pas strictement nécessaires : si on les enlève, on obtient 29 obligations, 27 sont prouvées par Simplify, 26 par Z3 et CVC3 et 22 par Alt-Ergo. Mais globalement toutes ces obligations sont prouvées par au moins un prouveur.

Par contre, le lemme `array_injective_is_surjective` n'est pas prouvé automatiquement, aussi pour compléter la preuve de ce programme, on passe alors en mode manuel : on utilise la sortie Coq de Why pour valider ce lemme. La preuve Coq complète est disponible sur le site de Capucine : `http://romain.bardou.fr/capucine`.

Voici les étapes de cette preuve. En premier lieu, un petit développement Coq indépendant est créé, avec le contenu suivant.

– Preuve dans $\mathbb{N}$ du théorème suivant : pour tout $n \in \mathbb{N}$ et pour toute fonction $f : \mathbb{N} \to \mathbb{N}$, si

1) $f$ envoie l'intervalle $[0; n-1]$ dans $[0; n-1]$ ;

2) $f$ est injective sur cet intervalle ;

alors $f$ est surjective : $\forall k \in [0; n-1], \exists j \in [0; n-1], k = f(j)$. Ce théorème est prouvé par récurrence sur $n$.

– Transposition de ce théorème pour une fonction de $\mathbb{Z}$ dans $\mathbb{Z}$.

Dans un second temps, on complète la preuve de l'énoncé Coq généré par Why à partir du lemme `injection_is_surjective`. Cela est très simple : il s'agit d'appliquer le théorème précédent pour la fonction $f(i) = select(a, i)$.

Donnons quelques détails sur la preuve du premier théorème ci-

dessus :

1) Le cas $n = 0$ est trivial.

2) Pour le cas $n = n' + 1$, on cherche à se ramener à une fonction sur l'intervalle $[0; n'[$. Pour cela, si on pose $k = f(n')$ , $t$ la transposition de $k$ et $n'$, et $g = t \circ f$. On montre successivement :

a) $g$ envoie l'intervalle $[0; n'[$ dans $[0; n'[$

b) $t$ est injective

c) $g$ est injective sur l'intervalle $[0; n'[$

d) $g$ est surjective (par hypothèse de récurrence)

e) $f = t \circ g$

f) et on conclut que $f$ est surjective

La preuve du même théorème mais pour une fonction $f$ sur $\mathbb{Z}$ consiste à se ramener au cas précédent en exhibant une fonction $g$ sur $\mathbb{N}$ qui coïncide avec $f$ sur $[0; n - 1]$. On prend par exemple $g(n) =$ `Zabs_nat`$(f($`Z_of_nat`$(n)))$. La preuve est ensuite un peu lourde mais directe, en jouant sur les conversions entre $\mathbb{N}$ et $\mathbb{Z}$.

## 5. Travaux proches, conclusions et perspectives

Pour cette étude de cas des tableaux creux, nous sommes partis d'une première spécification due à J.-C. Filliâtre et A. Paskevich, directement exprimée dans le langage de Why, et qui ne tenait pas compte des problèmes d'accès en dehors des bornes. Comme Why n'admet pas les alias, leur structure de tableaux creux est un $n$-uplet contenant des tableaux purs. Le problème d'alias potentiel est donc éliminé par construction de la modélisation. Nous avons alors adapté ces spécifications dans Capucine, contrôlé les alias avec les régions, ajouté les invariants pour garantir les accès des tableaux, et également effectué la preuve du lemme intermédiaire en Coq. De plus, dans leur modélisation l'invariant était un prédicat ajouté systématiquement en pré- et en post-condition des fonctions, là où nous n'avons à le prouver que lors de la mise à jour.

La méthode Capucine est inspirée de deux sortes d'approches : d'une part, les techniques de typage statique à base de régions et de capacités ;

et d'autre part les langages et systèmes logiques spécialisés pour raisonner sur les programmes avec partage. Les techniques de la première sorte visent à proposer des systèmes de types avancés pour contrôler les alias ou pour réaliser des politiques d'accès à des ressources. La notion de région dans les systèmes de types a été introduite by Jouvelot et Talpin en 1991 [TJ92] et utilisée pour la gestion de la mémoire [TT97]. La notion d'appartenance (*ownership*) est proposée par Clarke, Potter et Noble [CPN98] dans le but de contrôler le partage. Les notions de permissions ou de capacités sont introduites par Crary, Walker et Morrisett [CWM99] et les opérations d'adoption, de focus et d'unfocus par Fahndrich and Deline [FD02]. Toutes ces techniques permettent une gestion de la mémoire par un typage statique fin, mais n'ont pas été utilisées dans le but de faire de la preuve de propriétés fonctionnelles de programmes [CP08].

Au contraire, les approches de la seconde sorte sont considérées avec la preuve de programmes pour objectif. La logique séparante (*separation logic*) [Rey02] est une approche fructueuse, car elle intègre dans la logique les concepts de non-aliasing et de capacité d'accès. Néanmoins, il n'y aucun typage statique impliqué et toutes les difficultés sont transportées au niveau de la logique et donc déléguées aux prouveurs. Ceux-ci ne peuvent plus être des prouveurs standards, il faut des prouveurs qui connaissent les prédicats et connecteurs spécifiques de la logique séparante. Le concept *d'ownership*, dans le contexte de la vérification déductive, est proposée par Barnett et al. [BDF+04] en 2004. Elle permet pour la première fois une méthodologie correcte pour la préservation des invariants de données, même en présence d'appels réentrants que l'on peut rencontrer dans les langages à objets. Leur notion d'ownership est réalisée par des prédicats dans la logique, et non dans un typage statique. Dans cette approche, la relation d'appartenance est construite en déclarant quels sont les champs qu'une classe possède. Avec les régions de notre approche, les liens d'appartenance ne suivent pas forcément les pointeurs, comme montré par les exemples *observer pattern* et de tables de hachage de notre rapport de recherche [BM10]. Les types univers (*Universe types*) [DM05, MR07] ajoutent un niveau de typage statique à l'ownership mais ne permettent toujours pas de lien d'ownership différents des pointeurs. La *regional logic* [BNR08] permet des structures d'ownership plus générale, mais tout est traité dynamiquement : les ré-

gions sont des champs *ghost* que le programmeur doit mettre à jour manuellement. Les cadres dynamiques (*Dynamic frames*) [Kas06] permettent une approche similaire, en déclarant les régions comme des variables de spécification qui dénotent des empreintes mémoire. Comme l'ownership, les cadres dynamiques ne sont pas basés sur un typage statique, mais par contre apportent des avantages similaires à la logique séparante : les formules logiques contiennent des informations sur la structure de la mémoire, avec lesquelles on peut raisonner au niveau des prouveurs.

Notre objectif est de combiner les aspects précédents, en poussant le plus d'information possible du coté du système de types. C'est ce qui nous permet de générer les obligations de preuve uniquement là où c'est nécessaire. Par exemple les invariants ne sont à prouver qu'au moment de l'instruction `pack`. Notre système de type n'est pas aussi puissant que celui de Charguéraud et Pottier [CP08] : par exemple nous ne pouvons pas permettre les *strong update* qui autorisent le type d'un pointeur à changer lors de l'exécution. Par contre, nous pouvons déréférencer un pointeur dans une région possédée sans ouvrir son parent, ce qui est nécessaire pour pouvoir le lire dans les spécifications.

Comme travaux futurs, nous souhaitons naturellement appliquer l'approche Capucine aux autres exemples de la collection VACID. Ce sont d'autres exemples impliquant des invariants de données et du partage, donc de bons candidats pour notre approche. Une difficulté apparaît quand on ne peut pas séparer statiquement les pointeurs, car on doit alors raisonner dans un modèle mémoire analogue à un grand tableau unique. On aurait besoin de compléter l'approche statique de Capucine par des facilités plus dynamiques telles que celles apportées par la logique de séparation ou les cadres dynamiques.

Un objectif futur important est d'appliquer ces techniques à des programmes codés dans des langages comme Java ou C, par exemple en les traduisant vers le langage de Capucine. Cela implique d'ajouter des annotations de régions dans ces programmes, et un enjeu important est de limiter la quantité d'annotations à insérer manuellement.

*Remerciements.*

Nous remercions Jean-Christophe Filliâtre et Andrei Paskevich pour nous avoir permis de repartir de leurs spécifications initiales des tableaux creux en Why. Nous remercions également les évaluateurs anynomes pour leurs remarques constructives.

## Références

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Springer-Verlag, 2004.

[BDF+04] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6) :27–56, June 2004.

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software : The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer, 2007.

[BLS04] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# Programming System : An Overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2004.

[BM10] Romain Bardou and Claude Marché. Regions and permissions for verifying data invariants. Technical Report RR-7412, INRIA, 2010. `http://hal.inria.fr/inria-00525384/en/`.

[BNR08] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Regional logic for local reasoning about global invariants. In *European Conference on Object-Oriented Programming (ECOOP)*, Paphos, Cyprus, July 2008.

[Bor00] Richard Bornat. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction*, pages 102–126, 2000.

[CK04] David R. Cok and Joseph Kiniry. ESC/Java2 : Uniting ESC/Java and JML. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *CASSIS*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer, 2004.

[CP08] Arthur Charguéraud and François Pottier. Functional translation of a calculus of capabilities. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 213–224, September 2008.

[CPN98] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*, pages 48–64. ACM Press, 1998.

[CWM99] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 262–275. ACM Press, 1999.

[Dij76] Edsger W. Dijkstra. *A discipline of programming*. Series in Automatic Computation. Prentice Hall Int., 1976.

[DM05] Werner Dietl and Peter Müller. Universes : Lightweight ownership for JML. *Journal of Object Technology*, 4(8) :5–32, 2005.

[DMS+09] Markus Dahlweid, Michal Moskal, Thomas Santen, Stephan Tobies, and Wolfram Schulte. VCC : Contract-based modular verification of concurrent C. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Companion Volume*, pages 429–430. IEEE Comp. Soc. Press, 2009.

[FD02] Manuel Fahndrich and Robert Deline. Adoption and focus : practical linear types for imperative programming. In *Programming Language Design and Implementation (PLDI)*, volume 37, pages 13–24, May 2002.

[Fil03] Jean-Christophe Filliâtre. Verification of non-functional programs using interpretations in type theory. *Journal of Functional Programming*, 13(4) :709–745, July 2003.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.

[FM07] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177, Berlin, Germany, July 2007. Springer.

[Fra08] The Frama-C platform for static analysis of C programs, 2008. `http://www.frama-c.cea.fr/`.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10) :576–580 and 583, October 1969.

[Kas06] Ioannis T. Kassios. Dynamic frames : Support for framing, dependencies and sharing without restrictions. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *14th International Symposium on Formal Methods (FM'06)*, volume 4085 of *Lecture Notes in Computer Science*, pages 268–283, Hamilton, Canada, 2006.

[LLM07] Gary T. Leavens, K. Rustan M. Leino, and Peter Müller. Specification and verification challenges for sequential

object-oriented programs. *Formal Aspects of Computing*, 2007.

[LM10] K. Rustan M. Leino and Michał Moskal. VACID-0 : Verification of ample correctness of invariants of data-structures, edition 0. In *Proceedings of Tools and Experiments Workshop at VSTTE*, 2010.

[MR07] Peter Müller and Arsenii Rudich. Ownership transfer in universe types. In *ACM SIGPLAN conference on Object-oriented programming systems and applications (OOPSLA)*, pages 461–478. ACM, 2007.

[Rey02] J. C. Reynolds. Separation logic : a logic for shared mutable data structures. In *17h Annual IEEE Symposium on Logic in Computer Science*. IEEE Comp. Soc. Press, 2002.

[TJ92] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3) :245–271, 1992.

[TT97] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2) :109–176, 1997. Academic Press.