Université Paris-sud 11
INRIA Futurs
Équipe ProVal

ENS Cachan
MPRI

Rapport de stage

Invariants de classe et systèmes d'ownership

Romain Bardou

Mars – Juillet 2007

Stage réalisé sous la direction de Claude Marché

Table des matières

1 Introduction

1.1 L'INRIA Futurs

J'ai effectué mon stage à l'INRIA Futurs, dans l'équipe de recherche ProVal, sous la direction de Claude Marché. Le laboratoire se trouve sur le plateau de Saclay, à côté du Centre Scientifique d'Orsay de l'université Paris-sud 11. Cette équipe rassemble des chercheurs de l'INRIA, mais aussi des chercheurs du CNRS et des enseignants-chercheurs de l'université qui sont membres de l'équipe DEMONS du LRI (Laboratoire de Recherche en Informatique).

Une partie de l'équipe ProVal développe un ensemble d'outils dédiés à la preuve de programmes. Prouver un programme, c'est d'abord spécifier son comportement attendu. On veut alors prouver la correction du programme, c'est-à-dire montrer qu'il vérifie sa spécification. Éventuellement, on peut aussi vouloir montrer sa terminaison.

Les outils Caduceus[9] et Krakatoa[9] prennent des programmes C et Java, respectivement, en entrée. Ceux-ci sont annotés par des spécifications décrivant leur comportement attendu. Caduceus et Krakatoa les compilent en programmes Why. A l'aide de l'outil Why[9], on en extrait des obligations de preuve, c'est-à-dire des expressions logiques qu'il va falloir prouver. Ces expressions peuvent être données à des outils de démonstration automatique comme Simplify[4] ou Ergo[3], ou à des assistants de preuve comme Coq[13] (Figure 1).

Dans un souci de factoriser le travail effectué sur Caduceus et Krakatoa, une partie de l'équipe a commencé récemment à développer le langage et l'outil Jessie. Il s'agit d'un langage relativement proche du C, mais simplifié, avec des structures de données plus évoluées qu'en Why, en particulier des pointeurs. Jessie factorise certaines analyses que l'on voudrait faire à la fois en C et en Java. Pour prouver un programme C ou Java spécifié, on pourra d'abord le compiler en un programme Jessie, lui-même compilé en Why. Mon travail concerne cet outil, c'est-à-dire le langage Jessie et son interprétation en Why.

1.2 Invariants de structure et relation d'appartenance

Les outils de preuve de programmes orientés objets doivent fournir un langage de spécification suffisamment riche. Une notion importante est la notion d'invariant de programme ou de structure. Lorsqu'un programmeur déclare une variable ou définit une structure de données, il a une idée en tête : tel entier restera toujours positif, telle liste sera toujours triée, etc.

La correction du reste du code repose sur ces invariants. Par exemple, une fonction de recherche dichotomique dans un tableau suppose l'invariant : "le tableau est trié". Il faut vérifier que les fonctions modifiant le tableau, telles que les fonctions d'insertion ou de suppression, ne rompent pas cet invariant. Pour prouver la correction de son programme, le programmeur peut déclarer l'invariant "le tableau est trié". C'est alors le rôle des outils comme Jessie de générer des obligations de preuve telles que "la fonction d'insertion ne rompt pas l'invariant".

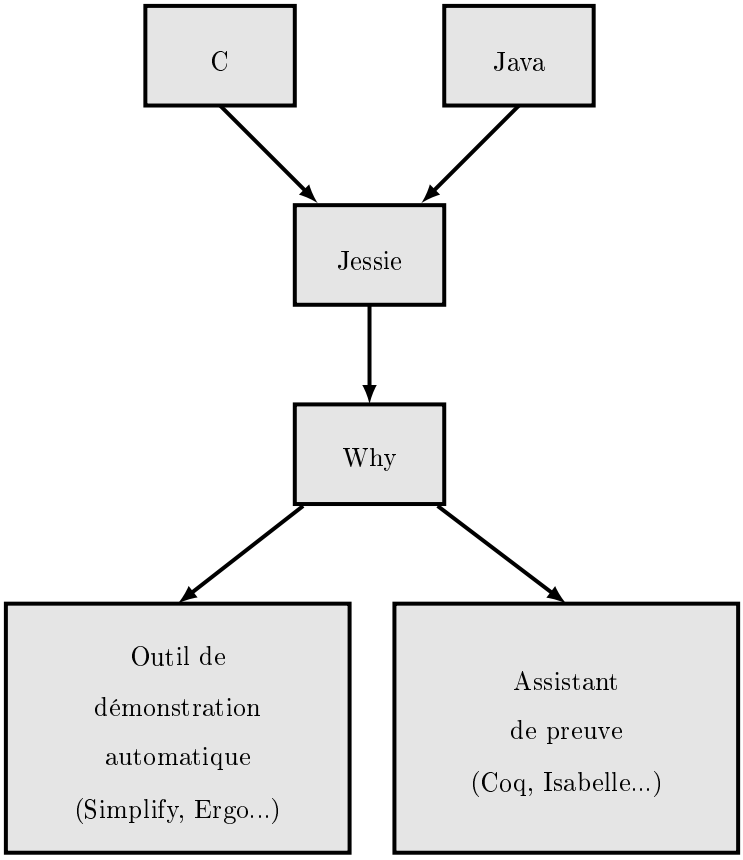

FIG. 1 – La plateforme Why

Mais il faut d'abord définir plus précisément la notion d'invariant. Est-ce qu'un invariant doit être vrai quel que soit le point de programme considéré ? C'est la définition la plus naturelle, mais elle pose un problème d'expressivité. Par exemple, une fonction d'insertion dans un tableau trié peut vouloir rompre temporairement l'invariant.

Un autre problème se pose. Imaginez un pointeur sur l'une des cases d'un tableau sensé rester trié en permanence. Modifier la valeur pointée par ce pointeur peut rompre l'invariant du tableau. Il s'agit d'un problème d'*aliasing* entre pointeurs. D'une manière générale, les cases du tableau pourraient être modifiées par une fonction sans même que celle-ci ait connaissance de l'invariant, qui est alors rompu.

Pour résoudre le problème de l'aliasing, des travaux ont proposé différentes notions d'*appartenance* entre objets. Spec# et son langage intermédiaire Boogie[1] utilisent un tel système. L'idée est qu'un objet est une boîte, fermée ou ouverte. Chaque boîte contient (possède) d'autres boîtes à l'intérieur, et une boîte fermée ne peut contenir que des boîtes fermées. Les invariants d'une boîte ne peuvent parler que des boîtes qu'elle contient, et si une boîte est ouverte, son invariant peut être rompu. Une boîte fermée ne peut pas être modifiée.

Dans l'exemple du tableau trié, le tableau est une boîte qui en contient d'autres (les cases). Pour modifier l'une des cases, il faut d'abord l'ouvrir, ce qui n'est possible que si le tableau est lui-même ouvert, et donc si on ne suppose pas son invariant. En refermant les boîtes, on doit montrer l'invariant.

Dans JML[2], Peter Müller propose une notion d'appartenance entre objets. Il introduit un système de type, appelé Universe Type System[5], permettant d'assurer statiquement qu'un objet ne modifie que les objets qu'il possède.

1.3 Contribution et plan

Mon stage a consisté à rechercher une notion d'invariant raisonnablement expressive et à l'implémenter dans Jessie. Au final, c'est une approche à la Boogie qui a été utilisée, après avoir été adaptée au modèle mémoire de Jessie. J'ai aussi effectué une réalisation en Coq de ce modèle mémoire.

Dans la section 2 nous verrons comment on spécifie un programme de façon générale, et comment ces spécifications deviennent des obligations de preuve. Ensuite dans la section 3 nous nous intéresserons plus particulièrement à Jessie et à sa traduction en Why. Nous verrons aussi ma première contribution : la réalisation en Coq du modèle mémoire de Jessie. Puis dans la section 4 nous verrons en détail les problèmes posés par les invariants, des exemples montrant leur utilité, et la façon dont ils sont traités dans Boogie. Enfin dans la section 5 nous verrons pourquoi la solution de Boogie ne convient pas directement et comment l'adapter pour Jessie, ce qui est la contribution principale de mon stage.

2 Spécifier et prouver un programme

Prenons un exemple simple : la fonction puissance, qui prend un réel x et un entier n en argument et qui renvoie x^n. L'algorithme utilisé est celui de la figure 2. On va montrer la correction de ce programme, c'est-à-dire le fait qu'il renvoie effectivement x^n.

Power(x, n) =
  $r \leftarrow 1$; $t \leftarrow x$; $i \leftarrow n$;
  Tant que $i > 0$:
    Si i est impair : $r \leftarrow r \times t$;
    $t \leftarrow t \times t$;
    $i \leftarrow \left[\frac{i}{2}\right]$;
  Fin
  Retourne r;
Fin

FIG. 2 – La fonction Power

2.1 Logique de Hoare

Historiquement, la logique de Hoare[10] permet de raisonner sur chaque instruction du programme. Cette logique est fondée sur les triplets de Hoare, notés $\{A\}I\{B\}$, où I est une instruction et où A et B sont des formules logiques. Un tel triplet est valide si I assure la post-condition B sous la pré-condition A.

En utilisant cette logique, on peut donner une *spécification* à notre programme Power, c'est-à-dire décrire son comportement. Le programme Power annoté par sa spécification est donné en figure 3.

Power(x, n) =
  $\{n \geq 0\}$
  $r \leftarrow 1$; $t \leftarrow x$; $i \leftarrow n$;
  Tant que $i > 0$:
    {Invariant : $r \times t^i = x^n$ et $i \geq 0$}
    Si i est impair : $r \leftarrow r \times t$;
    $t \leftarrow t \times t$;
    $i \leftarrow \left[\frac{i}{2}\right]$;
  Fin
  $\{r = x^n\}$
  Retourne r;
Fin

FIG. 3 – La fonction Power spécifiée

Cette spécification contient :

- la pré-condition de la fonction, $n \geq 0$, devant être valide à l'entrée du programme ;
- la post-condition de la fonction, $r = x^n$, devant être valide à la sortie du programme ;
- un invariant de boucle devant être valide à l'entrée de la boucle et à la fin de chaque itération.

2.2 Calcul de plus faible pré-condition

Dijkstra[6] a introduit le calcul de plus faible pré-condition, basé sur la logique de Hoare. L'idée est la suivante : on veut qu'une instruction I assure la post-condition B. On recherche la plus petite pré-condition W telle que le triplet de Hoare $\{W\}I\{B\}$ soit valide, c'est-à-dire telle que pour toute autre pré-condition A telle que $\{A\}I\{B\}$ soit valide, alors $A \Rightarrow W$.

Étant donné un triplet de Hoare, pour montrer sa validité il suffit donc de montrer que sa pré-condition A implique sa plus faible pré-condition W. Si W est calculable, on peut donc générer l'obligation de preuve $A \Rightarrow W$.

Notons que W n'est pas toujours calculable ; dans le cas des boucles par exemple, il est nécessaire de fournir des informations supplémentaires, en l'occurence un invariant de boucle. Celui-ci est une formule devant être valide à l'entrée de la boucle et après chaque itération. Si ces conditions sont vérifiées on peut en déduire que cet invariant est vrai à la sortie de la boucle.

L'outil Why utilise un calcul de plus faible pré-condition intégrant les post-conditions exceptionnelles, c'est-à-dire la possibilité pour une fonction de lever une exception et de spécifier, en plus du comportement normal, le comportement de la fonction lorsqu'elle lance une exception[8].

On peut donc donner le programme Power annoté par sa spécification à Why, qui nous donne alors un ensemble d'obligations de preuves. Entre autres, il faut prouver que l'invariant de boucle est valide. Sachant que $r \times t^i = x^n$, il faut montrer :

- $r \times t \times (t^2)^{[\frac{i}{2}]} = x^n$ quand i est impair ;
- $r \times (t^2)^{[\frac{i}{2}]} = x^n$ quand i est pair ;
- $\frac{i}{2} \geq 0$ quand $i > 0$.

Il faut aussi montrer la post-condition $r = x^n$. A la sortie de la boucle, on sait que son invariant est vérifié et que sa condition $i > 0$ est fausse. On en déduit que $i = 0$ et donc que $r = x^n$.

2.3 Prouver les obligations de preuve

L'outil Why peut générer les obligations de preuve dans plusieurs formats, adaptés à plusieurs outils de démonstration automatique ou à des assistants de preuve. Il suffit donc d'espérer que les outils de démonstration automatique prouveront le plus de buts possible, les buts restants étant prouvés à l'aide des assistants de preuve.

Il faut garder à l'esprit que l'on n'a prouvé que la spécification que l'on a donnée. Il faut donc bien choisir cette spécification si l'on veut se convaincre

que notre programme ne contient pas d'erreur.

3 Le langage et l'outil Jessie

3.1 Le langage Jessie

Jessie est un langage intermédiaire entre C et Java d'une part, et Why d'autre part. Il s'agit d'un langage impératif, possédant des types simples (entiers, booléens, réels) et des enregistrements. Les entiers sont des entiers mathématiques, c'est-à-dire non bornés. La syntaxe est proche de celle du C, et les programmes sont annotés par leur spécification.

3.1.1 Fonctions et comportements

On peut adapter l'exemple 3 à la syntaxe Jessie :

```
logic real lpower(real x, integer n)

real power(real x, integer n)
  requires n >= 0;
  behavior result_is_power:
    ensures \result == lpower(x, n);
{
  real r = 1.;
  real t = x;
  integer i = n;
  while (i > 0)
    invariant r * lpower(t, i) == lpower(x, n) && i >= 0;
    variant i;
  {
    if (i % 2 == 1) r = r * t;
    t = t * t;
    i = i / 2;
  }
  return r;
}
```

La pré-condition est donnée dans la clause `requires`, et la post-condition est donnée dans la clause `ensures`. La boucle est annotée par son invariant et par un `variant`. Celui-ci permet de donner une mesure permettant de prouver la terminaison de la boucle. Ici, la valeur de `i` décroît strictement à chaque itération sans pouvoir dépasser 0, donc on s'en sert comme variant.

Jessie permet de déclarer plusieurs comportements. Chaque comportement est donné dans une clause `behavior`. Ici, on a déclaré un unique comportement nommé `result_is_power`. Chaque comportement est prouvé indépendamment.

On a aussi dû déclarer la fonction logique `lpower` pour représenter les expressions de la forme x^n. Celles-ci n'existant pas en Jessie, il faut les définir. Nous reviendrons sur ce problème dans la section 3.2.2.

3.1.2 Enregistrements

Les enregistrements se déclarent de la façon suivante :

```
type Point = {
  integer x;
  integer y;
}
```

Dans cet exemple, on a déclaré un type `Point` contenant deux champs de type entier, `x` et `y`.

On peut étendre cet enregistrement :

```
type ColorPoint = Point with { integer color; }
```

On obtient donc un type `ColorPoint`, héritant des champs `x` et `y` du type `Point`, avec un champ supplémentaire nommé `color`. Cette définition induit une relation de sous-typage ; toute expression de type `ColorPoint` est aussi une expression de type `Point`.

On peut, dans les spécifications, savoir si une valeur est une instance d'un `Point` ou d'un `ColorPoint`. On écrit `x <: C` le prédicat "`x` est une instance de `C`".

On peut faire une coercion de toute expression `e` de type `Point`, en une expression de type `ColorPoint` en écrivant : `e :> ColorPoint`. Tout comme les "casts" en Java, ceci n'est pas sûr. En Jessie, chaque coercion génère une obligation de preuve. En écrivant `e :> ColorPoint` on génère l'obligation de preuve `e <: ColorPoint`, assurant ainsi que la coercion est correcte.

Pour finir, chaque variable dont le type est un enregistrement est en réalité un pointeur. Jessie propose une arithmétique de pointeurs : étant donnée une valeur `p` de type `Point` et un décalage `i`, on peut écrire le pointeur `p+i`. On peut voir `p` comme un tableau de `Point`, et `p+i` comme l'adresse de la i-ième case de ce tableau. Ceci a une conséquence importante : contrairement au langage C, Jessie ne permet pas à un pointeur, même décalé, de pointer au milieu d'une structure. Ces pointeurs ne sont pas pour autant toujours valides ; ils peuvent pointer sur une zone non allouée de la mémoire. Nous reviendrons là-dessus dans la section 3.4.

3.2 Le langage Why

3.2.1 Description du langage

Avant d'être transformés en obligations de preuve, les programmes Jessie sont compilés en Why. Il s'agit d'un langage fonctionnel, avec des effets de bord limités. En effet, Why possède des références à la Caml, mais à un seul niveau : une référence ne peut pas contenir une structure de donnée possédant elle-même une référence.

Un programme Why est constitué des constructions suivantes.

– Déclarations de types. Par exemple, pour déclarer le type `list` polymorphe :

```
type 'a list
```

– Déclarations de symboles logiques tels que `lpower`, que l'on a vu dans la section 3.1.1, ou `llength` qui à une liste associe sa longueur :

```
logic llength: 'a list -> int
```

– Définitions de prédicats. Le prédicat `empty` qui prend une liste en paramètre et qui est vrai si, et seulement si, cette liste est vide peut s'écrire ainsi :

```
predicate empty(l: 'a list) =
  llength(l) = 0
```

– Définitions d'axiomes. Par exemple, pour affirmer que la longueur d'une liste est toujours positive, on peut définir l'axiome `length_positive` :

```
axiom length_positive:
  forall l: 'a list. llength(l) >= 0
```

– Déclarations d'exceptions comme `Empty_list`.

```
exception Empty_list
```

– Déclaration de paramètres avec leur type, utilisables dans le corps des fonctions. Il s'agit de programmes déclarés en donnant leur type annoté, sans implémentation. Par exemple, on peut donner un paramètre `length` sensé calculer la longueur d'une liste, sans donner sa définition mais en spécifiant son comportement à l'aide d'un type annoté :

```
parameter length: l: 'a list ->
  { true }
  int
  { result = llength(l) }
```

– Définitions de fonctions. Par exemple, on peut écrire une fonction nommée `check_not_empty` qui lance une exception si son paramètre est une liste vide. Cette fonction est annotée avec une post-condition composée de deux cas : le cas normal, où la longueur de la liste est strictement positive, et le cas où l'exception `Empty_list` est levée, où la longueur de la liste est nulle.

```
let check_not_empty = fun (l: int list) ->
  if length(l) = 0 then
    raise Empty_list
  else void
  { llength(l) > 0 | Empty_list => llength(l) = 0 }
```

Les types peuvent être des types simples (`int`, `bool`...), des types fonctionnels (t_1 `->` t_2), ou des types annotés. Par exemple, `length` possède un type fonctionnel dont le résultat est un type annoté par la pré-condition `true` et une post-condition assurant que son résultat est cohérent avec `llength`.

A partir d'un tel programme, Why génère des obligations de preuves que l'on peut donner à de multiples outils de démonstration automatique ou assistants de preuve.

3.2.2 Prédicats et fonctions logiques

On a vu dans la section 3.1.1 qu'il fallait définir la fonction logique `lpower` pour représenter l'exponentielle. En réalité on ne définit pas cette fonction. On déclare simplement son existence :

```
logic real lpower(real x, integer n)
```

Il ne s'agit cependant que d'une déclaration. Nous verrons dans la section 3.3 comment donner le sens de `lpower`.

3.3 Techniques de preuve

Après avoir spécifié un programme, on se retrouve en général avec un certain nombre de fonctions logiques ou de prédicats sans définition, comme `lpower`. La façon de leur donner un sens dépend des outils utilisés pour prouver les obligations de preuve.

3.3.1 Assistant de preuve

Avec un assistant de preuve, la logique est suffisamment riche pour que l'on puisse facilement donner une définition aux fonctions. On peut ensuite appliquer cette définition pendant les preuves. Cette approche assure la cohérence du système, puisque l'on n'a pas besoin d'utiliser d'axiomes.

3.3.2 Outil de démonstration automatique

Les outils de démonstrations automatiques proposent des logiques moins expressives que celles des assistants de preuve. Il n'est pas évident de donner une définition à `lpower`, par exemple. Et si on y arrive, cette définition aura de grande chance d'être trop compliquée pour être utilisée par les outils de démonstration automatique.

Au lieu de donner la définition de `lpower`, on donne un ensemble d'axiomes décrivant son comportement. Par exemple, on a besoin de l'égalité suivante :

$$x \times \texttt{lpower}(x, n) = \texttt{lpower}(x, n + 1)$$

On peut donc définir un axiome Jessie nommé `lpower_mult` affirmant cette égalité pour tout x et n :

```
axiom lpower_mult: \forall real x; \forall integer n;
  x * lpower(x, n) == lpower(x, n+1)
```

3.3.3 Cohérence

Contrairement à l'approche basée sur des assistants de preuve, l'approche précédente, qui introduit de nouveaux axiomes, n'est pas sûre puisque ces axiomes peuvent rendre le système incohérent.

Cependant, on peut donner une réalisation de ces axiomes pour montrer leur cohérence. Par exemple, pour vérifier la cohérence de l'axiome `lpower_mult`, on

commence par donner une définition de `lpower` comme dans la section 3.3.1. Ensuite, on doit montrer l'axiome `lpower_mult` qui devient donc un lemme. En Coq, ce lemme s'écrit :

```
Lemma lpower_mult: forall x: R, forall n: Z,
  (x * lpower x n)%R = lpower x (n + 1).
```

J'ai appliqué cette approche pour montrer la cohérence du modèle mémoire de Jessie, qui est axiomatisé en Why. Je détaille cette réalisation dans la section 3.5.

3.4 Traduction de Jessie vers Why

3.4.1 Mémoires

Pour pouvoir raisonner sur les structures de données de Jessie, il faut d'abord avoir un modèle de la mémoire. On pourrait par exemple représenter la mémoire comme un grand tableau, dans lequel chaque donnée prend une certaine place. Mais pour obtenir des preuves plus simples on peut choisir un modèle de plus haut niveau.

En Jessie, les données que l'on manipule sont soit des types de base, soit des enregistrements sur lesquels on peut faire de l'arithmétique de pointeurs. On a vu dans la section 3.1.2 que lorsqu'on incrémente un pointeur, on ne peut pas tomber au milieu d'une structure ; ceci peut se refléter dans le modèle mémoire afin de simplifier les raisonnements. En particulier, comme un pointeur `p` ne peut pointer qu'au début d'une structure et que l'accès à un champ `f` ne peut se faire qu'en écrivant `p.f`, si on modifie `f` on est certain qu'on ne modifie pas un autre champ `g`. Cette propriété de *séparation* peut être obtenue gratuitement si on choisit correctement le modèle mémoire.

Le modèle choisi est donc le suivant. Soit s une structure d'enregistrement donnée. Un objet x de type s est un *pointeur*. Chaque champ f de s définit une *mémoire*, c'est-à-dire une fonction des pointeurs dans les valeurs.

Par exemple, la structure suivante :

```
type A = {
  integer value;
}
```

définit une mémoire `value`. Étant donné un pointeur `a` de type `A`, l'accès `a.value` correspond à appliquer la fonction `value` au pointeur `a`.

Concrètement, on définit en Why le type `'a pointer`, correspondant aux pointeurs sur des structures de type `'a`, et le type `('a, 'b) memory`, correspondant aux mémoires associant des valeurs de type `'b` aux pointeurs sur des structures de type `'a`. On définit ensuite les fonctions `select` et `store`. Le type de `select` est :

```
('a, 'b) memory, 'a pointer -> 'b
```

Son rôle est d'accéder à la valeur d'un champ. Par exemple, `select(value, a)` est une valeur de type entier correspondant à `a.value`. Le rôle de `store` est de modifier une mémoire ; il a le type suivant :

```
('a, 'b) memory, 'a pointer, 'b -> ('a, 'b) memory
```

Par exemple, si `value2` = `store(value, a, 5)`, alors la mémoire `value2` est égale à `value` sauf pour le pointeur `a` où elle vaut 5. D'une façon générale on a les axiomes suivant :

$$\forall m, p, x,\ \texttt{select(store(}m\texttt{, }p\texttt{, }x\texttt{), }p\texttt{)} = x$$

$$\forall m, p_1, p_2, x,\ p_1 \neq p_2 \Rightarrow \texttt{select(store(}m\texttt{, }p_1\texttt{, }x\texttt{), }p_2\texttt{)} = \texttt{select(}m\texttt{, }p_2\texttt{)}$$

Pour compiler la définition d'une structure, on déclare donc autant de mémoires qu'il y a de champs, avec le type `memory`. Mais ce type prend deux paramètres : le type du champ et le type de la structure, qu'il faut donc aussi déclarer. Ainsi, la structure `Point` devient, en Why :

```
type Point
parameter x : (Point, int) memory ref
parameter y : (Point, int) memory ref
```

Pour compiler l'accès à un champ, on utilise des paramètres Why : `acc_` et `upd_`, correspondant respectivement à la lecture et à la mise à jour d'une mémoire. Si `alloc` est la table d'allocation (voir la section 3.4.4), `p.x` est compilé en `acc_ alloc x p`. La post-condition de `acc_` est : `result = select(x, p)`. `p.x = e` est compilé en `upd_ alloc x p e`. La post-condition de `upd_` est : `result = store(x, p, e)`.

Pour compiler `ColorPoint`, on utilise le même type `Point`. En effet, tous les pointeurs de type `ColorPoint` seront aussi des pointeurs de type `Point`. Il suffit donc de rajouter les mémoires de `ColorPoint` :

```
parameter color : (Point, int) memory ref
```

3.4.2 Séparation des mémoires

Ce modèle a une conséquence importante : les mémoires sont *séparées*. Si on modifie `p.x`, on a modifié la mémoire `x` mais les mémoires `y` et `color` restent inchangées.

L'avantage de cette méthode est lorsqu'on modifie `p.x`, on garde toutes les propriétés qu'on a prouvées sur `y`. Dans un modèle sans mémoires séparées, il faudrait d'abord montrer que modifier `p.x` ne modifie pas le champ `y` (de `p` ou même d'autres pointeurs).

Cependant, on ne peut plus raisonner à propos de la mémoire *en un point de programme donné*. Ceci pose des problèmes pour exprimer les invariants. Par exemple, supposons que notre programme maintienne l'invariant suivant : "pour tout point `p`, `p.x`= 0 ou `p.y`= 0". Une façon naïve d'exprimer cet invariant serait d'écrire la propriété ainsi :

$$\forall p,\ p\texttt{.x} = 0 \vee p\texttt{.y} = 0$$

Mais dans cette expression, les mémoires `x` et `y` sont des variables libres. Si l'on quantifie sur `x` et `y` on obtient :

$$P : \forall p, x, y,\ p.x = 0 \vee p.y = 0$$

Mais cette propriété est différente. Par exemple, dans le programme suivant, qui vérifie bien la propriété attendue :

```
1: (* on suppose p.x = 1 et p.y = 0 *)
2: p.x = 0;
3: p.y = 1;
```

Au point `1` on a `p.x` $= 1$. Donc si x_1 est la mémoire `x` au point `1`, d'après P on a :

$$\forall y,\ p.x_1 = 0 \vee p.y = 0$$

c'est-à-dire :

$$\forall y,\ p.y = 0$$

Autrement dit, de P on peut déduire que `p.y` est toujours nul, ce qui est faux au point `3`. La section 4 sera consacrée à ce problème.

3.4.3 Hiérarchies de classes

Pour l'instant, rien ne différencie `Point` et `ColorPoint`, qui appartiennent à la même hiérarchie de classe (`ColorPoint` étant une extension de `Point`). En pratique, si `p` est un `Point`, on peut vouloir lire `p.color`. Pour cela il faut d'abord faire une coercion de `p` en un `ColorPoint`, et il faut donc pouvoir vérifier que `p` est aussi un `ColorPoint`.

Pour cela, on associe à chaque adresse mémoire un *tag* de type `T tag_id`, où `T` est le type associé à la hiérarchie, par exemple `Point`. Pour la hiérarchie `Point`, on définit les tags `Point_tag` et `ColorPoint_tag`, tous deux de type `Point tag_id`.

On déclare aussi une table associant leur tag à chaque pointeur, et des prédicats tels que `instanceof` pour lire cette table.

3.4.4 Table d'allocation et arithmétique de pointeurs

Les mémoires associent des valeurs à des pointeurs. Un pointeur est une adresse que l'on peut décaler. Pour savoir quels pointeurs sont valides, on utilise une *table d'allocation*. Celle-ci associe, à chaque adresse, un intervalle de décalages valides. On peut la voir comme un tableau (indexé par les adresses) de tableaux (indexé par les décalages).

Une mémoire peut aussi être vue comme un tableau à deux dimensions, où les cases non allouées ne sont pas définies.

La figure 4 montre un exemple de table d'allocation pour le type `Point`. Le pointeur d'adresse 2 et de décalage 1 représente un point `p1`, tel que `p1.x` $= 4$ et `p1.y` $= 2$.

Les fonctions `offset_max` et `offset_min` permettent de connaître l'intervalle des décalages valides que l'on peut appliquer à un pointeur. Un pointeur est valide si, et seulement si, `offset_min` ≤ 0 et `offset_max` ≥ 0, c'est-à-dire si le pointeur est valide sans décalage.

Décaler un pointeur est réalisé à l'aide de la fonction logique `shift`. Étant donné un pointeur `p` et un décalage `offset`, `shift(p, offset)` représente le pointeur `p` décalé de `offset`.

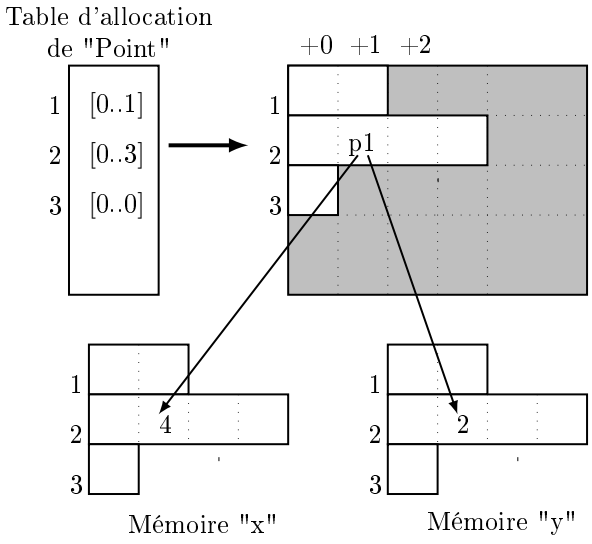

FIG. 4 – Table d'allocation et mémoires

Des axiomes permettent de relier `shift` avec `offset_max` et `offset_min` :

```
offset_max(a, shift(p, i)) = offset_max(a, p) - i
offset_min(a, shift(p, i)) = offset_min(a, p) - i
```

La variable `a` est la table d'allocation, `p` est un pointeur pour cette table d'allocation et `i` est un décalage.

3.5 Cohérence

Dans la section 3.4, on a vu que Jessie utilisait un certains nombres de types (`pointer`, `memory`...) et de fonctions logiques (`select`, `shift`...). Ceux-ci ne sont pas définis, mais des axiomes décrivent leur comportement. Pour garantir la cohérence de ce modèle, j'en ai fait une réalisation en Coq, en donnant des définitions aux types et aux fonctions logiques et en démontrant les axiomes à partir de ces définitions.

Le modèle mémoire a été défini en Why. On peut compiler cette définition pour obtenir un programme Coq à remplir : les déclarations deviennent des définitions vides[1], et les axiomes deviennent des lemmes à prouver.

Ces définitions sont relativement immédiates :

- L'ensemble des adresses est $\mathbb{N}$, l'ensemble des entiers naturels. Il s'agit des adresses utilisées par la librairie standard `Map`.
- Un pointeur est un élément de $\mathbb{N} \times \mathbb{Z}$, c'est-à-dire une paire constituée d'une adresse et d'un décalage.

[1] Pour pouvoir recompiler le modèle mémoire de Why à Coq sans perdre mon travail, j'ai dû remplir les définitions en mode preuve, avec la tactique `exact`.

```
Definition pointer: Set -> Set.
exact (fun _ => prod ad Z).
Defined.
```

Le paramètre, de type `Set`, correspond au type du pointeur. En pratique, la définition des pointeurs est indépendante de ce type. `ad` est le type des adresses, c'est-à-dire `N`.

– Une table d'allocation est une "map" qui, à chaque adresse, associe un entier relatif n représentant la longueur du tableau alloué à l'adresse donnée. Celui-ci sera valide pour les décalages de 0 à $n-1$. Le type Coq utilisé est celui de la librairie standard `Map`, et si aucun intervalle n'est associé à une adresse donnée, on suppose que cet intervalle est vide.

```
Definition alloc_table: Set -> Set.
exact (fun _ => Map Z).
Defined.
```

Comme pour les pointeurs, la définition des tables d'allocation ne dépend pas du type des éléments alloués.

– `offset_max` et `offset_min` sont calculés à partir de la deuxième composante d'un pointeur et de la longueur de l'intervalle donnée par la table d'allocation. `shift` se contente de modifier la seconde composante d'un pointeur.

```
(* Longueur du block alloué pour le pointeur p *)
Definition block_length (A: Set) (a: alloc_table A)
  (p: pointer A) :=
  match fst p with
    0%N => 0 (* le pointeur nul, jamais valide *)
  | _ =>
    match MapGet Z a (fst p) with
      None => 0
    | Some size => size
    end
  end.

Definition offset_max :
  forall (A1:Set), (alloc_table A1) -> (pointer A1) -> Z.
exact (fun A1 a p => block_length a p - snd p - 1).
Defined.

Definition offset_min :
  forall (A1:Set), (alloc_table A1) -> (pointer A1) -> Z.
exact (fun A1 t p => -snd p).
Defined.

Definition shift :
  forall (A1:Set), (pointer A1) -> Z -> (pointer A1).
exact (fun A1 p i => (fst p, snd p + i)).
Defined.
```

– Une mémoire doit associer une valeur à chaque décalage. Certains décalages n'étant pas valides, la mémoire a besoin d'une valeur par défaut,

notament pour pouvoir écrire les axiomes sur `store` et `select`. La librairie `Map` en Coq utilise pour clés des entiers positifs, mais un décalage peut être négatif. Une mémoire est donc une paire constituée d'une valeur par défaut et d'un contenu. Ce contenu est une `Map` associant à chaque adresse son intervalle de valeurs. Cet intervalle est représenté par deux `Map`, une pour les décalages positifs et une pour les décalages négatifs.

```
Definition block (T: Set) := prod (Map T) (Map T).
Definition blocks (T: Set) := Map (block T).

Definition memory: Set -> Set -> Set.
exact (fun S T => prod (blocks T) T).
Defined.
```

– Les définitions de `select` et `store` découlent naturellement de la définition des mémoires et des pointeurs. On commence par définir des fonctions pour lire et remplacer un bloc ou une case donnée d'une mémoire. Celles-ci ne présentent pas d'intérêt particulier ; elles se contentent d'appeler correctement les fonctions `MapGet` et `MapSet`.

A partir de ces définitions, on peut démontrer les axiomes de la section 3.4.

Écrire ces définitions n'est pas très difficile. Il faut savoir rechercher les lemmes de base pour la librairie `Map`. Je n'ai pas trouvé, dans la documentation de `Coq`, de librairie utilisant pour clés des entiers relatifs, ce qui aurait simplifié l'implémentation. Les preuves sont relativement mécaniques, mais il reste une difficulté importante à résoudre : elles doivent être constructives. Ceci amène parfois à chercher des preuves peu intuitives. Heureusement, le tiers-exclu ($A \vee \neg A$) est souvent prouvable lorsque A est une propriété sur des objets calculatoires tels que les entiers utilisés dans notre modèle.

Avoir réalisé ce modèle a permis de garantir la cohérence du modèle mémoire utilisé par les outils automatiques pour prouver des programmes Jessie, et donc aussi des programmes C et Java. Cela avait été fait pour Krakatoa, l'outil permettant de traduire de programmes Java en Why, mais pas pour Caduceus.

4 Les invariants

Dans cette section, nous présentons quelques motivations supplémentaires pour la notion d'invariant, ainsi que la solution adoptée par Boogie. Avant cela, il est utile de préciser la syntaxe que l'on va utiliser en Jessie pour déclarer un invariant :

```
type A = {
  integer i;
  invariant inv(this) = this.i > 0;
}
```

Ici, nous déclarons une structure `A` munie d'un champ `i` et d'un invariant nommé `inv`. Intuitivement, cet invariant affirme qu'un pointeur `this` sur une structure de type `A` possède un champ `i` strictement positif. Ici, `this` n'est pas un mot-clé : on peut utiliser n'importe quel identifiant, qui est alors lié dans le corps de

l'invariant.

4.1 Motivations

L'application la plus immédiate des invariants en Jessie est la traduction des invariants de classe en Java et de structure en C. D'autres fonctionalités réclament la présence d'invariants générés automatiquement (sections 4.1.1 et 4.1.2).

4.1.1 Taille des tableaux

Dans Jessie, si un champ d'une structure est lui-même de type structurel, il s'agit en réalité d'un pointeur. En tant que tel, il peut donc prendre un certain nombre de valeurs parmi une plage d'adresse. Cependant, seul un intervalle de cette plage est valide, et tout déréférencement du pointeur doit être accompagné d'une preuve de sa validité.

Par exemple, on peut définir un tableau d'entiers ainsi :

```
type Int = {
  integer value;
}

type IntArray = {
  Int[..] content;
  integer size;
}
```

La notation `[..]` signifie que le pointeur `content` n'est pas limité.

Un invariant pour la structure `IntArray` vient naturellement en lisant cette définition : le pointeur `content` doit être valide au moins dans l'intervalle 0 .. `size`−1. On pourrait donc écrire l'invariant suivant :

```
\offset_min(this.content) <= 0
  && \offset_max(this.content) >= this.size-1
```

4.1.2 Intervalle des pointeurs

Jessie permet de décrire l'intervalle valide d'un pointeur dans son type. Chaque type structurel doit être suivi d'un intervalle. Par exemple, l'intervalle `[..]` signifie que l'on n'a aucune contrainte sur `offset_min` et `offset_max`. L'intervalle `[a..b]` signifie que `offset_min` $\leq a$ et que `offset_max` $\geq b$. L'intervalle `[a]` signifie qu'un *offset* au moins est valide : a.

Ceci est pratique lorsqu'il s'agit du type d'un argument d'une function. Souvent, on utilise l'intervalle `[0]`, qui signifie simplement que le pointeur est valide sans décalage et qu'on peut donc le déréférencer directement. Ceci se traduit en une pré-condition sur la fonction. Mais lorsqu'il s'agit d'un type associé à la déclaration d'un champ, on a besoin de traduire l'intervalle en un invariant sur la structure.

Par exemple, on pourrait définir une structure `Point` ainsi :

```
type Point = {
  Int[0] x;
  Int[0] y;
}
```

Le but de l'intervalle `[0]` est de préciser que `x` et `y` ne sont pas des tableaux, mais des champs toujours valides. Cette définition de `Point` pourrait se traduire ainsi :

```
type Point = {
  Int[..] x;
  Int[..] y;
  invariant x_offset(this) =
    \offset_min(this.x) <= 0
    && \offset_max(this.x) >= 0
  invariant y_offset(this) =
    \offset_min(this.y) <= 0
    && \offset_max(this.y) >= 0
}
```

4.1.3 Union discriminées

Jessie est un langage intermédiaire entre C et Java d'une part, et Why d'autre part. Les unions de C peuvent se traduire en Jessie de façon automatique. Ceci a été traité par Yannick Moy en même temps que mon stage[11].

Considérons le programme C suivant :

```
struct shape {
  int discr;
  union {
    // Disque
    int radius;
    //@ invariant radius_positive(x) = x.radius >= 0;
    // Rectangle
    struct {
      int width;
      //@ invariant width_positive(x) = x.width >= 0;
      int height;
      //@ invariant height_positive(x) = x.height >= 0;
    } dim;
  } uni;
};

float area(struct shape *p) {
  switch (p->discr) {
    case 0: return 3.14 * p->uni.radius * p->uni.radius;
    case 1: return p->uni.dim.width * p->uni.dim.height;
  }
}
```

La structure `shape` est une *union discriminée*. Elle peut être un disque, avec un champ `radius`, ou un rectangle, avec un champ `dim`. Les champs `radius` et

`dim` se superposent : on ne peut pas utiliser les deux à la fois. Pour savoir si une valeur de type `shape` est un disque ou un rectangle, la structure `shape` possède un champ `discr` qui vaut 0 pour un cercle et 1 pour un rectangle. La fonction `area` renvoie l'aire d'une `shape` en fonction de sa forme (disque ou rectangle).

Pour exprimer le rôle du champ `discr`, on a besoin d'écrire un invariant sur la structure `shape`. En Jessie, celui-ci ressemble à :

```
invariant shape_inv(this) =
  (this.discr == 0 ==> this <: Disc)
  && (this.discr != 0 ==> this <: Rectangle)
```

On voit donc que les invariants en Jessie sont utiles non seulement pour traduire les invariants de l'utilisateur en C et en Java, mais aussi pour certaines opérations plus mécaniques.

4.2 Difficultés

Une possibilité serait d'utiliser la sémantique de JML. Pour chaque fonction, on rajoute en pré-condition et en post-condition la condition suivante : "tous les objets accessibles vérifient leurs invariants". On peut raffiner cette condition pour permettre à certaines fonctions de ne pas supposer ou maintenir les invariants de certains objets. Mais cette solution pose des problèmes importants.

4.2.1 Déterminer les objets accessibles

Le premier problème est le suivant : comment exprimer l'ensemble des objets accessibles ?

Il ne suffit pas de maintenir les invariants des paramètres des fonctions.

```
type A = {
  integer x;
  invariant inv_A(this) = this.x > 0;
}

type B = {
  A[0] a;
}

unit m(B[0] b) {
  b.a.x = 0;
}
```

La méthode `m` ne rompt pas l'invariant de son paramètre, `b`. Par contre, l'invariant de `b.a` est rompu.

4.2.2 Modularité : cacher les invariants

Une structure de donnée possède en général une interface qui masque certaines parties de son implémentation. Certains types peuvent être abstraits et certains champs peuvent être privés, par exemple.

Les invariants d'une structure doivent aussi pouvoir être cachés. D'abord, parce qu'ils peuvent parler de champs qui ne sont pas visibles de l'extérieur. D'autre part, certains invariants peuvent être maintenus par la structure pour des raisons spécifiques à l'implémentation choisie.

Par exemple, prenons une structure permettant de représenter un ensemble d'entiers. Une approche consiste à l'implémenter sous forme de liste, avec un invariant disant que chaque élément n'apparaît qu'une seule fois. Une autre approche consiste à utiliser un tableau de booléens, pour lequel cet invariant n'a plus de sens. Si on utilise une liste triée, peut-être que l'on souhaite pouvoir changer à tout moment l'ordre utilisé pour trier la liste. Pour cela, il faut le masquer à l'utilisateur.

Une bonne notion d'invariant devrait donc permettre de parler des invariants sans révéler leur contenu, afin de garder des interfaces suffisamment abstraites et donc de gagner en modularité.

4.3 La méthodologie de Boogie

4.3.1 Cas simple : champ `inv`

La méthodologie de Boogie pour les invariants est fondée sur les idées suivantes. On se place d'abord dans le cas où les invariants d'une classe ne parlent que des champs immédiats de cette classe. Une expression de la forme `x.f.g` n'est donc pas autorisée.

- Chaque objet possède un champ booléen `inv` indiquant si son invariant est vrai ou non.
- On ne peut pas modifier un objet si son champ `inv` est égal à `true`, parce que cela risquerait de rompre l'invariant. A chaque affectation de la forme `x.f = e`, on doit donc montrer que `x.inv = false`.
- On peut modifier ce champ `inv` à l'aide des directives `pack` et `unpack`. Pour marquer que l'invariant n'est plus forcément vérifié, on "ouvre" l'objet à l'aide de la directive `unpack`. Ceci a pour effet d'affecter la valeur `false` au champ `inv` de l'objet. Pour marquer que l'invariant est vérifié on le "ferme" à l'aide de la directive `pack`. Celui-ci génère une obligation de preuve : il faut démontrer l'invariant avant d'affecter `true` à `inv`.

Ceci permet d'obtenir la propriété suivante : à tout moment, pour tout objet `x`, si `x.inv` est égal à `true` alors l'invariant de `x` est vérifié.

4.3.2 Champ `committed`

La solution précédente limite l'expressivité des invariants. Pour pouvoir autoriser un invariant sur `x` à parler d'expressions de la forme `x.f.g`, il est nécessaire de s'assurer que l'on ne modifie pas le champ `x.f.g` tant que le champ `inv` de `x` est égal à `true`. Ceci n'est pas facile : comment savoir, en modifiant un champ `g` d'un objet `y`, que `y` n'est pas lui-même le champ `f` d'un objet `x`?

```
x.f.g = 1;
y = x.f;
...
```

```
/* arbitrairement plus loin dans le programme... */
y.g = 0;
return 100/x.f.g; /* division par 0 */
```

C'est ici qu'interviennent les idées d'appartenance entre objets. Pour qu'un invariant puisse parler de `x.f.g`, il est nécessaire que `x` *possède* son champ `f`, et que d'une façon ou d'une autre il ne soit pas possible de modifier un objet possédé par un autre directement.

La solution proposée pour Boogie consiste à rajouter, pour chaque objet, un champ booléen nommé `committed`. Celui-ci vaut `true` si l'objet est possédé par un autre. Lorsqu'on modifie un objet, on génère une obligation de preuve pour s'assurer que son champ `committed` est égal à `false`. On modifie ensuite `pack` et `unpack`.

- `pack x` vérifie d'abord que les champs de `x` ne sont pas déjà possédés par un autre objet, c'est-à-dire que leur champ `committed` est à `false`. Un objet ne peut être possédé que par un seul objet à la fois.
- `pack x` met ensuite à `true` le champ `committed` de chacun des composants de `x`, afin de marquer qu'ils appartiennent à `x`.
- `unpack x` change le champ `committed` des champs de `x` pour le mettre à `false`. En effet, en ouvrant `x`, il perd la propriété de ses champs.

On se rapproche ainsi de l'image des boîtes proposée dans la section 1.2. Notons que l'utilisateur peut spécifier, pour une classe donnée, les champs qui sont une *composante* de cette classe. Seuls ceux-ci pourront être possédés par les objets de cette classe, ce qui permet à l'invariant d'en parler. Dans ce rapport, pour simplifier, on suppose que tous les champs sont des composantes des classes dans lesquels ils sont définis.

4.3.3 Extension aux sous-classes

La méthodologie présentée jusqu'ici ne tient pas compte de l'existence de sous-classes. Celles-ci posent un problème important : lorsqu'on applique `pack` à un objet, il faut vérifier tous ses invariants. Par exemple, un `Point` pourra posséder non seulement les invariants de la classe `Point` mais aussi ceux de la classe `ColorPoint` s'il en fait effectivement partie. Cette information n'est pas disponible de façon statique.

Si l'on ne fait pas attention, on risque donc de rompre un invariant d'une sous-classe :

```
type Integer = {
  integer value;
}

type NotNull = Integer with {
  invariant not_null(this) = this.value != 0;
}

unit set_null(Integer[0] a)
  requires a.inv && not a.committed;
  behavior nullify:
```

```
    ensures a.inv && not a.committed;
{
  unpack(a);
  a.value = 0;
  pack(a);
}

integer error(NotNull[0] a)
  requires a.inv && not a.committed;
{
  set_null(a); /* invariant rompu */
  return 100/a.value; /* division par zéro */
}
```

La fonction `set_null` rompt l'invariant de son paramètre si celui-ci est un `NotNull`, ce qui ne se voit pas à première vue puisque le type statique du paramètre est `Integer`. Si on ne fait pas attention, `pack(a)` va donc fermer `a`, indiquant ainsi que son invariant est vrai alors que ce n'est pas le cas.

Pour résoudre ce problème, on modifie le champ `inv` pour qu'il contienne non pas un booléen mais une classe. Il s'agit de la sous-classe la plus précise pour laquelle l'invariant est vrai. On indique à l'aide d'un paramètre supplémentaire à `pack` et `unpack` la classe vers laquelle, ou à partir de laquelle, on souhaite fermer ou ouvrir l'objet. `pack` et `unpack` sont modifiés en conséquence. En particulier, `pack` ne vérifie que les invariants de la classe vers laquelle on ferme l'objet, qui ne sont pas déjà connus grâce au champ `inv`.

Par exemple, supposons que l'on ait un objet x de la classe `ColorPoint`. On souhaite utiliser `pack` pour modifier x.`inv` en `ColorPoint`. Pour cela, `pack` impose de vérifier que x vérifie les invariants de la classe `ColorPoint`, mais aussi ceux de `Point`. Donc `pack` impose que :

- x.`inv` = `Point`, ce qui implique que x vérifie l'invariant de `Point`,
- et que x vérifie l'invariant de `ColorPoint`.

En résumé, les définitions de `pack` et `unpack` dans Boogie sont données dans la figure 5. S est la super-classe immédiate de T, $type(p)$ est le type dynamique de p, $Comp_T(x)$ est l'ensemble des composants de x définis dans la sous-classe T, et $Inv_T(x)$ est la conjonction des invariants définis dans T appliqués à x.

5 Adapter la méthodologie de Boogie à Jessie

5.1 Difficultés

On ne peut pas implémenter directement la méthodologie de Boogie dans Jessie. Le premier problème vient de la séparation des mémoires. Comme on l'a vu dans la section 3.4.2, on ne peut pas fournir d'axiome reliant deux mémoires différentes pour tout point de programme.

Or, l'intérêt de la méthodologie de Boogie est qu'elle assure un certain nombre de propriétés, en particulier : en tout point de programme, si le champ

`pack` x `as` $T =$
  Vérifier $x \neq null \wedge x.inv = S$;
  Vérifier $Inv_T(x)$;
  Pour tout $p \in Comp_T(x)$
    Vérifier $p = null \vee (p.inv = type(p) \wedge \neg p.committed)$;
    Si $p \neq null : p.committed \leftarrow true$;
  Fin
  $x.inv \leftarrow T$;

`unpack` x `from` $T =$
  Vérifier $x \neq null \wedge x.inv = T \wedge \neg x.committed$;
  $x.inv \leftarrow S$;
  Pour tout $p \in Comp_T(x)$
    Si $p \neq null : p.committed \leftarrow false$;
  Fin

FIG. 5 – `pack` et `unpack` dans Boogie

`inv` d'un objet est égal à une classe `C`, alors les invariants de `C` et de ses super-classes sont vrais en ce point de programme pour cet objet. Pour utiliser cette propriété, on voudrait en faire un axiome de la forme :

$$\forall x,\ x.inv <: C \Rightarrow Inv_C(x)$$

où $Inv_C(x)$ est l'invariant pour la classe C appliqué à l'objet x. Le symbole `<:` signifie habituellement "est une instance de" ; on l'utilise ici pour comparer deux classes. T `<:` S signifie donc "S est une super-classe de T".

Comme on l'a vu dans la section 3.4.2, à partir du moment où l'invariant pour la structure C mentionne autre chose que des constantes, cet axiome est incohérent dans Jessie. Et sans cet axiome, on perd tout l'intérêt des invariants.

5.2 Solution proposée

Afin de résoudre ce problème, on peut soit chercher une notion d'invariants différente, soit chercher un moyen pour parler des mémoires en un point de programme donné. C'est finalement cette dernière solution que j'ai choisie.

5.2.1 Prédicat `assoc`

Intuitivement, on doit pouvoir relier ensemble les mémoires correspondant à un point de programme donné. Ensuite, les axiomes seront quantifiés sur des mémoires reliées entre elles.

Une première idée consiste à rajouter un prédicat `assoc(pp, m)` signifiant intuitivement que la mémoire `m` correspond au point de programme `pp`. Notons

qu'une même mémoire peut être associée à plusieurs points de programmes successifs, si aucune affectation la modifiant n'a lieu.

On choisit de représenter les points de programme par des entiers. On rajoute donc une déclaration Why de `assoc` dans le modèle de Jessie :

```
logic assoc: int -> ('a, 'b) memory -> prop
```

Il reste maintenant à construire ce prédicat. Sa construction dépend du programme Jessie considéré, donc elle doit être faite pendant la compilation de Jessie vers Why.

A chaque point de programme, on insère une *boîte noire*. Une boîte noire en Why est une instruction dont on oublie l'implémentation. On lui ne lui donne rien d'autre qu'une spécification, comportant entre autres une post-condition. Cette post-condition sera supposée vraie dans la suite du programme. On peut donc l'utiliser pour construire le prédicat `assoc`.

Par exemple, si l'on a une structure de type `Point` avec deux mémoires `x` et `y` que l'on veut relier au point de programme 5, on utilise une boîte noire avec comme post-condition :

$$\texttt{assoc}(5, \texttt{x}) \wedge \texttt{assoc}(5, \texttt{y})$$

Pour bien comprendre comment sont compilés les points de programme, prenons un exemple complet :

```
type Point = {
  integer x;
  integer y;
}

unit move(Point[0] this, integer mx, integer my) {
  this.x = this.x + mx;
  this.y = this.y + my;
}
```

Une fois compilée en Why et en simplifiant la syntaxe, la fonction `move` ressemble à :

```
let move =
  fun (this: Point pointer) ->
  fun (mx: int) ->
  fun (my: int) ->
    (* boîte noire pour le point de programme 1 *)
    [ assoc(1, x) and assoc(1, y) ];

    (* première affectation *)
    x := store(x, this, select(x, this) + mx);

    (* boîte noire pour le point de programme 2 *)
    [ assoc(2, x) and assoc(2, y) ];

    (* deuxième affectation *)
    y := store(y, this, select(y, this) + my);
```

```
(* boîte noire pour le point de programme 3 *)
[ assoc(3, x) and assoc(3, y) ]
```

Une fois ce programme compilé par Why, on se retrouve, à la fin de la fonction, avec 4 mémoires : `x`, `y`, `x0` et `y0`, et avec les hypothèses suivantes sur ces mémoires :

```
x0 = store(x, this, select(x, this) + mx)
y0 = store(y, this, select(y, this) + my)
assoc(1, x)
assoc(1, y)
assoc(2, x0)
assoc(2, y)
assoc(3, x0)
assoc(3, y0)
```

5.2.2 Axiomes

Maintenant qu'on peut exprimer le fait que des mémoires correspondent à un même point de programme, on peut écrire les axiomes dont on a besoin. Il s'agit en fait d'une adaptation des propriétés prouvées pour Boogie[1].

On a besoin du prédicat Inv_T, pour une structure T donnée. Il s'agit de la conjonction des invariants définis par le programmeur dans T. En général, il dépend de l'état des champs de l'objet.

On note $\mathcal{M}_{f_1 \cdots f_i}$ l'ensemble des tuples de mémoires $(m_1, \cdots, m_i)$ possibles pour les champs $f_1 \cdots f_i$. Si les invariants de T mentionnent les champs $f_1 \cdots f_i$ (`inv` et `committed` inclus), on note $\mathcal{M}_T = \mathcal{M}_{f_1 \cdots f_i}$. On note $\forall M \in \mathcal{M}_T$ pour $\forall m_1 \cdots \forall m_i$.

Étant donné $M \in \mathcal{M}_T$, on pourra donc écrire $Inv_T(x, M)$ pour signifier que les invariants de x pour la structure T sont vrais pour les mémoires M.

On note $Assoc(pp, M) = \bigwedge_{j=1}^{i} assoc(pp, m_j)$. Intuitivement, $Assoc(pp, M)$ est vrai si M forme un ensemble cohérent de mémoires au point de programme pp.

On peut maintenant écrire les axiomes dont on a besoin[2].

- Le champ `inv` permet de connaître l'état des invariants d'un objet. On note $\texttt{inv}_M$ la composante de M correspondant à `inv`.

$$\forall pp, x, T,\ \forall M \in \mathcal{M}_T,\ Assoc(pp, M) \Rightarrow x.\texttt{inv}_M \texttt{ <: } T \Rightarrow Inv_T(x, M)$$

- Un objet possédé par un autre est forcément entièrement "fermé", c'est-à-dire que tous ses invariants sont vérifiés.

$$\forall pp, x,\ \forall M \in \mathcal{M}_{\texttt{inv},\texttt{committed}},\ Assoc(pp, M) \Rightarrow x.\texttt{committed}_M \Rightarrow x.\texttt{inv}_M = Type(x)$$

[2] En réalité, ces axiomes doivent aussi être quantifié sur la table des étiquettes de type.

$Type(x)$ est le type dynamique de x. Dans cet axiome, les seules mémoires qui apparaissent sont `inv` et `committed` ; on n'a donc besoin de quantifier que sur les couples de mémoires `inv` et `committed` reliées entre elles par `assoc`.

- Une boîte fermée ne contient que des boîtes fermées. On note $Comp_T$ l'ensemble des champs définis dans la structure T.

$$\forall pp, x, T,\ \forall M \in \mathcal{M}_{\texttt{inv,committed}},\ Assoc(pp, M) \Rightarrow \\ x.\texttt{inv}_M \texttt{ <: } T \Rightarrow \forall f \in Comp_T,\ x.f.\texttt{committed}_M$$

- Un objet n'a qu'un seul possesseur valide ("fermé") à la fois.

$$\forall pp, x, T, x', T', y,\ \forall M \in \mathcal{M}_{\texttt{inv,committed}},\ Assoc(pp, M) \Rightarrow \\ \left.\begin{array}{r} y \in Comp_T(x) \cup Comp_{T'}(x') \\ y.\texttt{committed} \\ x.\texttt{inv} \texttt{ <: } T \\ x'.\texttt{inv} \texttt{ <: } T' \end{array}\right\} \Rightarrow x = x'$$

Le premier axiome est le plus important ; il permet de déduire les invariants d'un objet à partir de l'état de son champ `inv`. Les deux axiomes suivant sont utiles pour simplifier l'écriture des pré-condition et des post-condition des fonctions. Au lieu de donner l'état de chaque champ un par un, il suffit de dire qu'un paramètre de la fonction est "fermé" pour la structure T (c'est-à-dire que son champ `inv` est une instance de T) pour en déduire l'état des champs `committed` et `inv` de tous ses champs définis dans T.

5.2.3 Optimisation du nombre de `assoc`

La solution basée sur le prédicat `assoc` présente un inconvénient évident : il faut générer beaucoup de "boîtes noires", ce qui ajoute beaucoup d'hypothèses dans les preuves. La plupart de ces hypothèses sont inutiles, surtout si l'on n'utilise pas les invariants. Pour réduire le nombre de ces hypothèses, on ne va générer les boîtes noires que dans les cas susceptibles d'être utiles pour les invariants.

Au début d'une fonction (au premier point de programme) on n'associe entre elles que les mémoires qui apparaissent dans la définition d'un des invariants d'un des paramètres de la fonction.

- Étant donné un des paramètres de la fonction, on considère les invariants de toute sa hiérarchie. Par exemple, si une fonction possède un paramètre de type `Point`, on pourrait avoir besoin d'utiliser les invariants déclarés dans `ColorPoint` s'il y en a.
- On considère aussi les invariants de tous les champs accessibles à partir des paramètres, c'est-à-dire toutes les expressions de la forme $x.f_1.\cdots.f_k$, éventuellement avec des coercions.

Ensuite, à chaque affectation d'un champ `f`, on crée un point de programme où l'on génère des hypothèses sur `assoc`. Seule la mémoire `f` étant modifiée, il est suffisant de ne générer que les hypothèses nécessaires pour associer les mémoires apparaissant dans les invariants dépendant de `f`.

Ceci limite le nombre d'hypothèses sur `assoc` générées. En particulier, si l'on n'utilise pas d'invariants, on ne relie que les mémoires au début des fonctions.

5.2.4 Se passer de `assoc`

En pratique, on peut se passer du prédicat `assoc`. A la place, on peut utiliser les boîtes noires pour générer les axiomes instanciés par les mémoires courantes. En effet, le prédicat `assoc` ne sert qu'à utiliser ces axiomes. Avec cette méthode on peut donc se passer de ceux-ci.

Ceci a plusieurs avantages. D'abord, au lieu d'avoir une liste d'instances de `assoc` en hypothèses lorsqu'on fait une preuve, on a directement les axiomes, ce qui est plus clair. De plus, les axiomes et les boîtes noires sont des sources d'incohérences ; en supprimant une partie de ceux-ci le système devient plus sûr.

Concrètement, si un invariant `i` d'une structure S, dépendant de la mémoire `m`, risque d'être brisé (par exemple lors d'une modification de `m`), au lieu de générer l'hypothèse :

$$assoc(pp, m) \wedge assoc(pp, inv) \wedge assoc(pp, committed)$$

on génère directement l'hypothèse suivante :

```
global_invariant_S(m, inv, committed)
```

où `global_invariant_S` est un prédicat paramétré par les mémoires `m`, `inv` et `committed` et qui représente les axiomes pour la structure `S` instanciés pour ces mémoires. Il ressemble à :

$$\texttt{global\_invariant\_S}(M) =$$

$$\forall x, valid(x) \Rightarrow \begin{cases} \forall T,\ x.\texttt{inv}_M \texttt{ <: } T \Rightarrow Inv_T(x, M) \\ x.\texttt{committed}_M \Rightarrow x.\texttt{inv}_M = Type(x) \\ \forall T,\ x.\texttt{inv}_M \texttt{ <: } T \Rightarrow \forall f \in Comp_T,\ x.f.\texttt{committed}_M \end{cases}$$

Le terme $valid(x)$ exprime la validité du pointeur x, c'est-à-dire le fait qu'il pointe sur une structure effectivement allouée. En particulier, x est non nul.

5.2.5 `pack` et `unpack` dans Jessie

Dans Boogie, `pack` et `unpack` sont implémentés. Les conditions sur les champs `inv` et `committed`, ainsi que sur l'invariant du paramètre dans le cas de `pack`, sont implémentées par des assertions : l'instruction "`assert a`" impose que l'expression booléenne `a` soit évaluée à `true`. La modification des champs `inv` et `committed` est implémentée par des affectations.

Dans Jessie, `pack` et `unpack` sont générés, pour chaque structure, sous la forme de paramètres Why. Étant donné qu'un paramètre Why n'a pas d'implémentation, on ne peut utiliser ni les assertions, ni les affectations.

On encode les assertions comme des pré-conditions, et les affectations par des post-conditions. Par exemple, pour encoder une unique affectation de la forme :

```
x.f = 0;
```

on utilise la post-condition suivante, où `f@` est la mémoire `f` avant l'application du paramètre :

```
f = store(f@, x, 0)
```

5.2.6 Cohérence

On a vu que l'on pouvait se passer du prédicat `assoc`, ce qui nous permet de nous passer des axiomes et donc de se convaincre plus facilement de la sûreté de l'implémentation. Il reste à justifier la cohérence des boîtes noires que l'on génère.

Pour cela, on réutilise les preuves qui ont été faites pour Boogie[1]. On en déduit qu'en tout point de programme, les propriétés des champs `committed` et `inv` (c'est-à-dire le prédicat `global_invariant_S`) sont vérifiées. On peut donc les fournir au programmeur.

5.2.7 Tableaux

En Jessie, on se sert des pointeurs et des décalages de pointeurs pour représenter les tableaux. Par exemple, un champ :

```
Point[..] t
```

représente un tableau de `Point`. Pour accéder à la i-ème case de `t`, on écrit simplement `t+i`.

Dans le cadre des invariants à la Boogie, cela signifie qu'un champ tel que `p` peut représenter un nombre arbitraire de composants (c'est-à-dire de "sous-boîtes") pour la classe dans laquelle il est défini : une par case allouée du tableau.

Tels qu'ils ont été présentés, `pack` et `unpack` ne lisent et ne modifient que la première case de chaque champ, c'est-à-dire sans décaler le pointeur. Ceci signifie que seule cette case peut profiter des invariants. Si un objet `x` possède un champ `f`, seule la case `x.f` est possédée par `x`. Ceci a deux conséquences importantes sur les autres cases, de la forme `x.f+i`, avec `i` $\neq 0$.

- Leur invariant peut être rompu alors que `x` est fermé. Au moment où on ouvre `x`, on ne peut pas supposer que ces cases vérifient leur invariant.
- Les invariants de `x` ne peuvent pas parler de ces cases que `x` ne possède pas. Par exemple, dans la structure `PointArray`, l'invariant `diag` est invalide :

  ```
  type PointArray = {
    Point[..] t;
    invariant diag(this) = \forall integer i,
      \valid(this.t+i) ==> (this.t+i).x = (this.t+i).y;
  }
  ```

  Le terme `\valid(p)` signifie que le pointeur `p` est alloué. On l'utilise ici pour ne quantifier que sur les décalages de `this.t` qui correspondent à une case valide du tableau.

Pour étendre les invariants aux tableaux de Jessie, on modifie `pack` et `unpack` : $Comp_T(x)$ fait référence à tous les décalages de tous les champs de x.

Ceci est suffisant pour autoriser l'invariant de `PointArray`. Lorsque l'on applique `pack` sur un objet `x` de type `PointArray`, on devra montrer que non seulement `x.f`, mais aussi tous ses décalages valides, sont entièrement fermés.

Pour modifier les cases d'un tableau de `Point`, il faut d'abord leur appliquer `unpack`. Pour éviter au programmeur d'avoir à écrire une boucle qui ouvre chacune des cases du tableau une par une, on introduit les paramètres `pack_block` et `unpack_block`, qui prennent un pointeur, une classe et deux entiers a et b en entrée et qui sont l'équivalent de `pack` et `unpack` sur tous les décalages $a, a+1, \cdots, b$ de ce pointeur.

5.3 Expressivité des invariants

Avant de décider d'adapter Boogie à Jessie, j'ai essayé d'exprimer les invariants sous forme de prédicat inductif. L'idée était ensuite de permettre au programmeur d'utiliser ce prédicat dans les pré-conditions et les post-conditions. L'avantage immédiat par rapport à Boogie est que la séparation des mémoires ne pose pas de problème particulier. Dans cette section, je décris ce prédicat inductif avant de comparer son expressivité avec celle des invariants de Boogie.

Pour simplifier, on se place dans le cas où il n'y a pas d'héritage ou de tableaux. En particulier, le champ `inv` est un booléen, vrai si l'invariant est vérifié.

5.3.1 Définition des invariants à l'aide de prédicats inductifs

Dans Boogie, quand l'invariant d'un objet est vérifié, les composants de l'objet doivent eux-mêmes vérifier leurs invariants. Autrement dit, pour fermer une boîte, ses sous-boîtes doivent déjà être fermées. Là-dessus se rajoute une relation d'appartenance entre les objets, c'est-à-dire les boîtes : une boîte est possédée par celle, unique, qui la contient.

On peut exprimer ces deux notions à l'aide d'un prédicat construit inductivement sur les composants d'un objet, en partant de la boîte la plus à l'intérieur. Ce prédicat I s'écrit de la façon suivante, pour x de type T.

$$I_T(x) = valid(x) \Rightarrow (Inv_T(x) \wedge \forall f \in Comp_T,\ I_{type(f)}(x.f))$$

Comme précédemment, $Inv_T(x)$ est l'invariant défini par l'utilisateur pour la structure T appliqué à x. $Comp_T$ est l'ensemble des champs définis dans T, et $type(f)$ est le type du champ f.

Ce prédicat est défini en fonction de lui-même. On peut, étant donnée la liste des structures du programme considéré, le définir en Coq à l'aide d'une définition inductive, même dans le cas où une structure T possède un champ lui-même de type T.

Pour prouver une expression de la forme $I_T(x)$, on doit commencer par montrer I pour tous les champs f définis dans T. Ceci implique que l'objet x forme un arbre. Pour montrer $I_T(x)$, il faut d'abord commencer par montrer le

prédicat I pour les feuilles de l'arbre ; ensuite on peut remonter progressivement à $I_T(x)$ en montrant l'invariant de chacun des nœuds de l'arbre. Une feuille dans cet arbre est le pointeur *null* ou tout autre pointeur invalide.

5.3.2 Expressivité

On a l'implication suivante :

$$\forall x \in T,\ x.\texttt{inv} \Rightarrow I_T(x)$$

En effet, si une boîte x est fermée (c'est-à-dire si $x.\texttt{inv} = \texttt{true}$), alors par construction :

– son invariant $Inv_T(x)$ est vérifié,

– et ses champs, s'ils sont valides, sont aussi fermés.

Ceci correspond exactement à la définition de $I_T(x)$.

Ceci a une conséquence intéressante. Si l'on a construit, en Boogie ou en Jessie, à l'aide d'utilisations successives de `pack` et de `unpack`, un objet x avec $x.\texttt{inv} = \texttt{true}$, alors on a $I_T(x)$ et donc x forme un arbre. Cet arbre peut être vu comme "l'arbre des boîtes" et correspond à la relation d'appartenance entre objets. On retrouve donc la propriété d'unicité du possesseur d'un objet, que l'on a utilisée comme axiome.

5.4 Exemple

Afin de voir comment on peut utiliser les invariants en pratique, regardons l'exemple de la figure 6. Le type `Hero` représente le héros d'une aventure, qui peut éventuellement être un sorcier (sous-structure `Sorcerer`), et qui se déplace avec une épée (structure `Sword`). La fonction `attack` permet d'exécuter une attaque d'un héros sur un autre.

Toutes les obligations de preuve sont prouvées automatiquement par Simplify ou Ergo, sauf une, la plus importante : la fonction `attack` doit maintenir l'invariant des héros, qui est que leur vie est toujours entre 0 et 100, et qu'ils sont morts si leur vie est 0.

On peut le montrer en Coq. On sait grâce à la pré-condition `target.inv = Hero` que le héros `target` a une vie entre 0 et 100 au départ, à laquelle on retranche les dégats de l'épée. On peut prouver que ceux-ci sont strictement positifs : on déduit de la pré-condition `this.inv = \typeof(this)` et de `global_invariant` que `this.committed` est vrai, donc que `this.sword.inv` est une sous-classe de `Sword`, et donc que `this.sword` vérifie l'invariant `damage_inv`. Donc après le coup d'épée la vie du héros est inférieure à 100. Le test qui suit permet d'assurer que cette vie reste positive et que si elle est nulle, le héros est bien mort.

Cet exemple montre une limitation de la méthode : on peut passer un `Sorcerer` à `attack` pour le paramètre `target`, mais à condition qu'il soit déjà à moitié ouvert : son champ `inv` doit être égal à `Hero`. Pour résoudre ce problème, on voudrait modifier la pré-condition de `attack` pour que `inv` puisse être n'importe quelle sous-classe de `Hero`. Mais `unpack(target)` réclame que `target.inv` soit exactement égal à `Hero`. Il faut donc, à l'aide de l'opérateur "`<:`", tester

```
type Sword = {
  integer damage;
  /* une épée fait des dégats strictement positifs */
  invariant damage_inv(this) = this.damage > 0;
}

type Hero = {
  integer life;
  boolean dead;
  rep Sword[0] sword; /* rep: sword est un composant de Hero */
  /* la vie d'un héros est entre 0 et 100,
     et si elle vaut 0, le héros est mort */
  invariant life_inv(this) =
    (this.life >= 0) &&
    (this.life <= 100) &&
    (this.life == 0 ==> this.dead);
  /* l'épée est valide (traduction du [0] devant son type) */
  invariant sword_inv(this) =
    \offset_min(this.sword) <= 0 &&
    \offset_max(this.sword) >= 0;
}

type Sorcerer = Hero with {
  integer mana;
  /* un sorcier possède de la mana, au moins 0 */
  invariant mana_inv(this) = this.mana >= 0;
}

/* un héros attaque un autre héros avec son épée */
unit attack(Hero[0] this, Hero[0] target)
  requires
    this.inv == \typeof(this) &&
    target.inv == Hero &&
    target.committed == false;
{
  /* on change target.inv en bottom (super-classe de Hero) */
  unpack(target);
  /* on sait que this.sword est valide,
     grâce à l'invariant sword_inv */
  target.life = target.life - this.sword.damage;
  if (target.life <= 0) {
    target.life = 0;
    target.dead = true;
  }
  /* on change target.inv en Hero */
  pack(target, Hero);
}
```

FIG. 6 – Exemple `game.jc`

le type dynamique de `target` pour éventuellement effectuer `unpack(target, Sorcerer)` avant `unpack(target)`.

6 Conclusion

La méthodologie de Boogie n'est pas applicable directement dans Jessie, du fait de la séparation des mémoires. Celle-ci empêche les propriétés démontrées pour Boogie, indispensables pour utiliser les invariants, d'être posées comme axiomes pour les programmes Jessie.

J'ai donc cherché un moyen de contourner le problème, en changeant complètement de méthodologie ou en reliant les mémoires séparées en chaque point de programme. C'est cette dernière solution que j'ai implémentée dans Jessie. On obtient un système aussi expressif que celui de Boogie décrit dans l'article de Barnett et al.[1], adapté à Jessie.

6.1 Travaux reliés

Le problème principal lorsque l'on veut maintenir un invariant pour une structure est d'établir quels sont les moyens pour le rompre. Ensuite, on cherche à restreindre ces moyens le moins possible mais suffisamment pour pouvoir fournir une notion d'invariant.

La méthode que j'ai implémentée est adaptée de celle utilisée dans Boogie[1]. Elle consiste à rajouter des champs aux objets indiquant si on suppose que l'invariant est vérifié, et à n'autoriser les affectations que dans le cas contraire. Cette méthode introduit aussi une notion de *composants* pour les objets. Un champ composant d'un objet est un champ possédé, c'est-à-dire que seul cet objet pourra modifier ce champ.

Cette notion d'appartenance entre objet se retrouve dans JML[5]. Il s'agit d'un système de type, appelé Universes, où les types objets sont annotés de façon à savoir par qui ils sont possédés et donc si on peut les modifier. Ceci pourrait être utilisé dans le cadre des invariants.

D'autres systèmes considèrent plus généralement le problème des alias entre pointeurs. On peut citer entre autres les analyses basées sur la logique de séparation[12].

6.2 Travaux futurs

La méthodologie de Boogie est un cadre de travail assez général que l'on peut adapter. Certains choix peuvent être faits pour simplifier les annotations de fonctions. Par exemple, une fonction aura souvent besoin de l'invariant de ses paramètres. Dans le cadre de Boogie, cela signifie que l'on pourrait supposer que leurs champs `inv` sont à `true`. Par contre, cela impose d'appliquer `unpack` avant de modifier ces paramètres. Il pourrait être intéressant d'étudier quelles hypothèses faire par défaut sur les paramètres des fonctions, voire s'il est possible d'inférer en partie les hypothèses les mieux adaptées.

L'exemple de la section 5.4 a montré une des limites de `pack` et `unpack` : on ouvre ou on ferme les boîtes une classe à la fois. Une façon de résoudre le problème serait de modifier `unpack` pour remplacer le test $x.inv = T$ par $x.inv$ `<:` T. On peut le faire aussi pour `pack`, mais il faut vérifier les invariants de toutes sous-classes qui ne sont pas des super-classes de `inv`. Avant d'implémenter cette solution il faut vérifier qu'elle est cohérente. Notons que dans Boogie, il existe une autre extension de `pack` et `unpack` visant à résoudre le problème, mais elle semble plus restrictive.

Plus généralement, en cherchant à faire de la preuve de programmes on se rend compte que l'on a souvent besoin de montrer que deux pointeurs sont différents. En particulier, si l'on a besoin d'une propriété sur un pointeur que l'on a montrée au début d'un programme, il faut montrer que toutes les affectations que l'on a faites depuis n'ont pas modifié ce pointeur. Les systèmes d'appartenance entre objets résolvent ce problème en partie seulement.

Je remercie Claude Marché pour son encadrement. Je remercie aussi Christine Paulin pour avoir bien voulu relire ce rapport, et tous les membres du laboratoire pour leur accueil et l'aide qu'il m'ont apportée pendant mon stage.

Références

[1] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6) :27–56, June 2004.

[2] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 2004.

[3] S. Conchon and E. Contejean. The Ergo automatic theorem prover. `http://ergo.lri.fr/`.

[4] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify : a theorem prover for program checking. *J. ACM*, 52(3) :365–473, 2005.

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

[6] E. W. Dijkstra. Notes on structured programming. In O. Dahl, E. Dijkstra, and C. Hoare, editors, *Structured programming*. Academic Press, 1971.

[7] J.-C. Filliâtre. *Preuve de programmes impératifs en théorie des types*. Thèse de doctorat, Université Paris-Sud, July 1999.

[8] J.-C. Filliâtre. Verification of non-functional programs using interpretations in type theory. *Journal of Functional Programming*, 13(4) :709–745, July 2003. [English translation of [7]].

[9] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In W. Damm and H. Hermanns, editors, *19th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science, Berlin, Germany, July 2007. Springer.

[10] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10) :576–580 and 583, Oct. 1969.

[11] Y. Moy. Union and cast in deductive verification. In *Proceedings of the C/C++ Verification Workshop*, volume ICIS-R07015. Radboud University Nijmegen, July 2007. `http://www.lri.fr/~moy/union_and_cast/union_and_cast.pdf`.

[12] 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.

[13] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.1*, July 2006. `http://coq.inria.fr`.