Analyse de Bytecode par Raffinement

Boubacar Demba Sall, Frédéric Peschanski, Emmanuel Chailloux

Analyse de Bytecode par Raffinement

Boubacar Demba Sall, Frédéric Peschanski, Emmanuel Chailloux
Sorbonne Universités, UPMC Univ Paris 06, CNRS, LIP6 UMR 7606
4 place Jussieu 75005 Paris, France.
{boubacar.sall | frederic.peschanski | emmanuel.chailloux}@lip6.fr

Résumé

La technique du *raffinement* permet la dérivation de programmes corrects par construction à partir des spécifications. Dans cet article nous proposons une application de cette technique formelle à l'analyse des programmes pour un langage de bas niveau de type *bytecode*. Il s'agit plus précisément de s'appuyer sur l'assistant de preuve *Coq* pour formaliser un analyseur de bytecode, et prouver sa correction.

1. Introduction

Dans cet article, nous nous intéressons à la preuve formelle de correction d'une analyse de programmes dans le système *Coq* [18]. Certains concepts manipulés en analyse de programmes (sémantique, domaine abstrait, invariant, garde, pré/post condition, ...) n'étant pas toujours présents dans le formalisme natif des assistants de preuve, un encodage de ces concepts peut s'avérer nécessaire. Dans le cas d'un *plongement profond* (*deep embedding*), l'introduction de nouvelles structures de données permet de représenter les concepts requis. On trouve dans [5] un plongement profond de la théorie de l'*interprétation abstraite* [8] qui permet de construire une analyse statique certifiée dans le système *Coq*. Dans le cas d'un *plongement léger* (*shallow embedding*), on s'efforce de se limiter au formalisme disponible pour exprimer les concepts requis. Comparé au plongement profond, un plongement léger est moins expressif mais permet de réduire les niveaux d'indirection qui complexifient le raisonnement. Le *raffinement* [19, 11, 10, 4] est une démarche d'abstraction qui a été bien étudiée, et qui semble se prêter à un plongement léger dans un formalisme issu de la théorie des types. On trouve dans [6] et [14] des exemples de plongements légers du raffinement permettant la dérivation de programmes corrects par construction dans le système *Coq*.

Nous proposons de privilégier un plongement léger, et d'appliquer la démarche de raffinement à la preuve de correction d'une analyse de programmes pour un langage de bas niveau de type *bytecode*. Dans la section 2, nous présentons le *raffinement de données* en faisant ressortir sa relation avec l'analyse de programmes. La section 3 présente les étapes d'un plongement léger dans le système *Coq* d'une analyse de bytecode, et décrit l'encodage des obligations de preuve (à satisfaire) permettant de valider la correction de l'analyseur. La section 4 présente quelques travaux connexes et la section 5 conclut cet article.

2. Raffinement de données et analyse de programmes

Soit $T = (\pi s, os_1, os_2, ..., os_n)$ une signature comprenant la signature πs d'un constructeur qui permet de créer un objet de type T, et les signatures os_i d'opérations qui agissent sur les objets de type T. Soient deux types abstraits de données $A = (\pi a, oa_1, oa_2, ..., oa_n)$ et $C = (\pi c, oc_1, oc_2, ..., oc_n)$ ayant la même signature T. Dans un souci de simplification nous nous limiterons aux opérations séquentielles et déterministes ($\pi a, oa_i, \pi c$ et oc_i sont des fonctions). On dit que C raffine A si pour tout programme $P[T]$ paramétré par leur signature commune, on a tout comportement observable de $P[C]$ est un comportement observable de $P[A]$ et $P[C]$ termine [1] à chaque fois que $P[A]$ termine. Pour prouver que C raffine A, il faut établir une correspondance entre leurs espaces d'état respectifs. Cette correspondance peut être spécifiée par un *invariant de liaison* $I(a, c)$ qui formalise le lien logique qui existe entre un état abstrait a et un état concret c [12].

Cas des opérations gardées. Dans le domaine des systèmes réactifs, les opérations sont liées à des évènements. L'applicabilité des opérations est généralement spécifiée par une garde : condition nécessaire à la survenue de l'évènement associé (ce dernier est réputé impossible dès lors que la garde correspondante est fausse, donc dans ce cas l'opération laisse l'état inchangé). On notera par Go_i la garde associée à l'opération o_i. Dans un tel contexte, pour prouver que C raffine A en s'appuyant sur l'invariant de liaison I, il suffit de réunir les conditions suivantes :

- *Etablissement de l'invariant* : $I(\ \pi a(...),\ \pi c(...)\ )$
- *Maintien de l'invariant* : $\forall\ i\ a\ c, I(a, c) \wedge Goc_i(c)\ \Rightarrow\ I(oa_i(a), oc_i(c))$
- *Renforcement des gardes* : $\forall\ i\ a\ c, I(a, c) \wedge Goc_i(c)\ \Rightarrow\ Goa_i(a)$

Il s'agit des obligations de preuve permettant de valider le raffinement de données dans les systèmes réactifs [3] en général, et entre machines *Event-B* [1] en particulier.

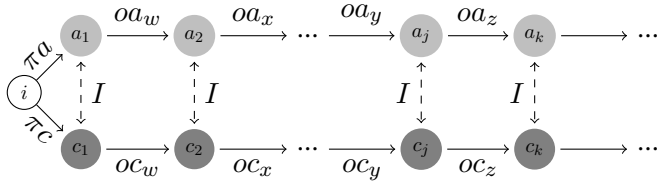
Cas des opérations partielles. Dans le cadre classique de la programmation séquentielle le domaine d'une opération est spécifié par une précondition : condition suffisante à une exécution sans erreur de l'opération (une violation de la précondition mène à un résultat indéfini). On notera par Po_i la précondition associée à l'opération o_i. Prouver que C raffine A dans ce cadre requiert les conditions suivantes :

- *Etablissement de l'invariant* : $I(\ \pi a(...),\ \pi c(...)\ )$
- *Maintien de l'invariant* : $\forall\ i\ a\ c, I(a, c) \wedge Poa_i(a)\ \Rightarrow\ I(oa_i(a), oc_i(c))$
- *Affaiblissement des préconditions* : $\forall\ i\ a\ c, I(a, c) \wedge Poa_i(a)\ \Rightarrow\ Poc_i(c)$

Il s'agit des obligations de preuve permettant de valider le raffinement entre modèles *Z*[16] ou *B classique*[2].

1. ne boucle pas indéfiniment, et s'arrête normalement (pas d'erreur à l'exécution)

Du raffinement de données à l'analyse de programmes. Comme l'illustre la figure ci-dessous, les obligations de preuve énoncées précédemment, assurent qu'à toute trace d'exécution $\tau_c = (c_1, c_2, ..., c_n)$ de $P[C]$ est associée une trace de même cardinalité $\tau_a = (a_1, a_2, ..., a_n)$ de $P[A]$ telle que $\forall\ j \cdot I(a_j, c_j)$.



Donc, lorsque C raffine A, pour tout programme P, et tout état concret c_j atteignable [2] par $P[C]$, il existe un état abstrait a_j atteignable par $P[A]$ tel que $I(a_j, c_j)$ est vrai. Cette correspondance entre les états atteignables de $P[C]$ et ceux de $P[A]$ peut être exploitée à des fins de vérification. En particulier, si tous les états atteignables a de $P[A]$ sont tels que $Poa_i(a)$ est vrai à chaque fois que oa_i est exécutée dans l'état a, alors la condition d'*affaiblissement des préconditions* nous garantit que tous les états atteignables c de $P[C]$ sont tels que $Poc_i(c)$ est vrai à chaque fois que oc_i est exécutée dans l'état c. Il s'agit d'une propriété intéressante car elle garantit qu'aucune opération ne sera jamais activée en dehors de son domaine de définition. La relation de raffinement entre A et C permet de déduire cette propriété de sûreté pour $P[C]$ à partir du moment où on a réussi à établir cette même propriété pour $P[A]$. En d'autres termes, si C raffine A, alors analyser $P[A]$ permet de conclure à propos de $P[C]$.

Pour concevoir une analyse, il peut être intéressant de s'appuyer sur des propriétés supposées établies (grâce par exemple à une autre analyse). Prenons l'exemple du langage de bytecode de la *JVM*, et de l'instruction *getfield f* qui permet de charger le champ *f* d'un objet à partir d'une référence. Durant l'exécution d'un programme bien formé, à chaque fois que l'instruction *getfield* est sur le point d'être exécutée, nous pouvons compter sur le fait que la référence de l'objet est présente en tête de la pile car la *JVM* effectue une analyse permettant de s'en assurer. Ceci peut être exprimé plus formellement en considérant que cette instruction admet la *garde* suivante : *length* (*stk c*) ≥ 1, où *c* représente un état concret et *stk* est un accesseur qui retourne la pile des opérandes. Par contre, nous ne pouvons pas toujours compter sur le fait que cette référence est différente de *null*, alors que dans ce cas l'opération n'est pas définie (elle provoque une erreur d'exécution). On considérera donc que cette instruction requiert la précondition suivante : *top* (*stk c*) $\neq$ *null*. Nous utiliserons des gardes pour représenter les propriétés supposées établies, et des préconditions pour représenter les propriétés à établir. Il nous faut donc considérer le raffinement dans le cas des opérations partielles et gardées.

Raffinement des opérations partielles et gardées. Lorsqu'on associe à une opération à la fois une garde et une précondition, nous l'interpréterons comme suit : (1) l'opération est définie si $Go_i \Rightarrow Po_i$, (2) si l'opération s'exécute c'est que Go_i

2. un état ϕ est atteignable par P ssi il existe une trace $\tau = (..., \phi, ...)$ admissible par P

est vrai au début de l'exécution, (3) l'opération est indéfinie si $Go_i \wedge \neg Po_i$. Cette interprétation est tirée de [15], où des conditions suffisantes de raffinement sont aussi proposées. Cependant le traitement est limité au raffinement algorithmique (sans changement de représentation des données).

Pour étudier la situation dans le contexte du raffinement de données, plaçons nous d'abord dans le cas des opérations gardées, et observons ensuite ce qui se passe lorsque les opérations sont rendues partielles. Supposons que C raffine A et considérons un programme P. Soit une trace $\tau_a = (..., a_j, a_k, ...)$ admissible par $P[A]$ en correspondance avec une trace $\tau_c = (..., c_j, c_k, ...)$ admissible par $P[C]$. Si maintenant on rend les oc_i partielles en supprimant c_j de leur domaine (pour ce faire il suffit d'associer les préconditions appropriées à ces opérations), la trace τ_c n'est plus admissible par $P[C]$. La relation de raffinement peut perdurer tant que d'autres traces correspondent encore à τ_a. Mais quand τ_c est la seule trace en correspondance avec τ_a, C ne raffine plus A. Cette situation est celle que permet d'éviter l'obligation d'*affaiblissement des préconditions* dans le cas des opérations partielles. Ainsi en fusionnant les obligations de preuve des deux cas précédents, et en apportant une légère modification à l'obligation de *maintien de l'invariant* (pour tenir compte de la partialité des opérations), on obtient les conditions suffisantes pour prouver que C raffine A dans un contexte où les opérations peuvent être partielles et gardées :

- *Etablissement de l'invariant* : $I(\ \pi a(...),\ \pi c(...)\ )$
- *Maintien de l'invariant* : $\forall\ i\ a\ c, I(a,c) \wedge Goc_i(c) \wedge Poa_i(a)\ \Rightarrow\ I(oa_i(a), oc_i(c))$
- *Renforcement des gardes* : $\forall\ i\ a\ c, I(a,c) \wedge Goc_i(c)\ \Rightarrow\ Goa_i(a)$
- *Affaiblissement des préconditions* : $\forall\ i\ a\ c, Goc_i(c) \wedge Poa_i(a) \wedge I(a,c)\ \Rightarrow\ Poc_i(c)$

3. Formalisation et preuve d'une analyse de bytecode en *Coq*

Soit un programme P constitué d'une liste d'instructions bytecode. Soit S_c la sémantique opérationnelle associée au langage de bytecode en question. On peut considérer que P est paramétré par le jeu d'instructions du langage (la signature de S_c). Une analyse de programme pour le langage de bytecode peut alors être spécifiée sous la forme d'une sémantique abstraite S_a ayant la même signature que S_c et décrivant de manière opérationnelle le fonctionnement de l'analyseur. Si S_c raffine S_a alors l'analyse est correcte dans le sens où, les propriétés de sûreté établies pour $P[S_a]$ sont valables pour $P[S_c]$. On peut donc en déduire la démarche suivante pour prouver la correction d'une analyse de programmes étant donnée la sémantique concrète S_c du langage de programmation : il suffit (1) de formaliser S_c et S_a, (2) de formaliser l'invariant de liaison qui relie les deux sémantiques, et (3) de satisfaire aux obligations de preuve énoncées plus haut afin d'établir que S_c raffine S_a.

Syntaxe et environnements d'exécution. Pour formaliser les sémantiques, nous commençons par spécifier la syntaxe des instructions du langage, ensuite on décrit les environnements d'exécution concret et abstrait, ainsi que les environnements

d'exécution initiaux. Par exemple, dans l'extrait ci-dessous on s'intéresse au langage de bytecode de la *JVM* et à la détection des dé-référencements de pointeurs *null*. On déclare donc le type *Instr* dont chacun des constructeurs représente une instruction du langage. On déclare ensuite les types *ConcreteState* et *AbstractState* qui représentent respectivement les environnements d'exécution concret et abstrait. On définit enfin *ConcreteInit* et *AbstractInit* qui représentent respectivement les environnements initiaux concret et abstrait.

Inductive *Instr* := *new* | *aload v* | *astore v* | *getfield f* | *setfield f* | *ifnonnull* | ...

Environnement concret	Environnement abstrait
Record *ConcreteState* := *mkCS* {	***Record*** *AbstractState* := *mkAS* {
stk : list Ref ;	*stk : list AbstractRef ;*
heap : Ref → Field → Ref	*nonnull : list AbstractRef*
}.	}.
Definition *ConcreteInit* := *mkCS [] (fun ...).*	***Definition*** *AbstractInit* := *mkAS [] [].*

AbstractState représente l'environnement d'exécution du futur analyseur. Le champ *nonnull* est une liste des références symboliques (abstraites) pour lesquelles les références concrètes correspondantes ont été identifiées comme différentes de *null*. Cette liste pourra être mise à jour à la création d'une nouvelle référence ou suite à un test de nullité, l'analyseur pourra ainsi s'assurer de la non nullité d'une référence en testant sa présence dans cette liste.

Invariant de liaison. La définition de l'invariant de liaison va consister à fournir une fonction qui construit un prédicat reliant un état abstrait a à un état concret c. Dans l'exemple ci-dessous, l'invariant de liaison stipule que dans les deux contextes d'exécution la pile des opérandes contient le même nombre d'éléments, et que si une référence a été identifiée dans l'abstraction comme n'étant pas *null*, alors son vis-à-vis dans le contexte d'exécution concret n'est effectivement pas *null*.

Definition *Invariant (a : AbstractState) (c : ConcreteState) : Prop* :=
length (stk a) = length (stk c) ∧
∀ v_a v_c, *In* (v_a,v_c) *(combine* [3]*(stk a) (stk c))* ∧ *In* v_a *(nonnull s)* → $v_c \neq$ *null*

Sémantiques des instructions. Pour spécifier le fonctionnement opérationnel de chaque instruction, on définira les fonctions *Guard*, *Pre* et *Exec* préfixées de *Concrete* ou *Abstract* selon la sémantique concernée. Dans la continuité des exemples précédents, l'extrait ci-dessous illustre une définition des sémantiques concrète et abstraite d'instructions bytecode. Les paramètres *i*, *c* et *a* sont respectivement de type *Instr*, *ConcreteState* et *AbstractState*. Les fonctions sont définies par filtrage de l'argument *i*. *ConcreteGuard* et *AbstractGuard* spécifient les gardes pour chaque instruction. *ConcretePre* et *AbstractPre* spécifient les préconditions associées à chaque instruction.

3. *combine : (list A) → (list C) → list (A×C)*
(combine $[v_{a1}; v_{a2}; ...; v_{an}]$ $[v_{c1}; v_{c2}; ...; v_{cn}]) = [(v_{a1}, v_{c1}); (v_{a2}, v_{c2}); ...; (v_{an}, v_{cn})]$

Sémantique concrète	Sémantique abstraite
Definition *ConcreteGuard i c : Prop :=* ***match*** *i* ***with*** *| getfield f ⇒ (length (stk c)) ≥ 1* *| ...* ***end***. ***Definition*** *ConcretePre i c : Prop :=* ***match*** *i* ***with*** *| getfield f ⇒ (top (stk c)) ≠ null* *| ...* ***end***. ***Definition*** *ConcreteExec i c* *: option ConcreteState :=* ***match*** *i,(stk c)* ***with*** *| getfield f, r::s ⇒* *Some (mkCS ((heap c r f)::s) (heap c))* *| ...* ***end***.	***Definition*** *AbstractGuard i a : Prop :=* ***match*** *i* ***with*** *| getfield f ⇒ (length (stk a)) ≥ 1* *| ...* ***end***. ***Definition*** *AbstractPre i a : Prop :=* ***match*** *i* ***with*** *| getfield f ⇒ In (top (stk a)) (nonnull a)* *| ...* ***end***. ***Definition*** *AbstractExec i a* *: option AbstractState :=* ***match*** *i,(stk a)* ***with*** *| ifnonnull, r::s ⇒* *Some (mkAS s (r::(nonnull a)))* *| ...* ***end***.

ConcreteExec et *AbstractExec* sont partielles (d'où le type *option* dans leur déclaration), elles retournent *None* lorsque la garde ou la précondition est fausse, dans le cas contraire elles calculent une mise à jour de l'environnement d'exécution correspondant.

Obligations de preuve. Afin de prouver la correction de l'analyse, nous devons maintenant énoncer les obligations de preuve nous permettant de conclure que la sémantique concrète raffine la sémantique abstraite :

Lemma *Initiation : Invariant AbstractInit ConcreteInit.*

Lemma *Invariance :* $\forall\ i\ a\ c$, *Invariant* $a\ c \wedge$ *ConcreteGuard* $i\ c \wedge$ *AbstractPre* $i\ a$ $\rightarrow \forall\ a'\ c'$, *AbstractExec* $i\ a$ = *Some* $a' \rightarrow$ *ConcreteExec* $i\ c$ = *Some* $c' \rightarrow$ *Invariant* $a'\ c'$.

Lemma *GuardStrengthening :* $\forall\ i\ a\ c$, *Invariant* $a\ c \wedge$ *ConcreteGuard* $i\ c \rightarrow$ *AbstractGuard* $i\ a$.

Lemma *PreconditionWeakening :*
$\forall\ i\ a\ c$, *ConcreteGuard* $i\ c \wedge$ *AbstractPre* $i\ a \wedge$ *Invariant* $a\ c \rightarrow$ *ConcretePre* $i\ c$.

Il nous reste ensuite à établir la validité des lemmes énoncés ci-dessus en nous appuyant sur les tactiques disponibles dans le système *Coq*. Une fois cette étape effectuée, notre analyse est correcte vis-à-vis du langage cible tel que spécifié par la sémantique concrète.

4. Travaux connexes

Les méthodes *Z* et *B* sont basées sur le raffinement de données mais diffèrent de l'approche présentée ici. En particulier, ces méthodes, comme les plongements en *Coq* du raffinement qu'on trouve dans [6, 7, 9], donnent une interprétation différente à la combinaison garde/précondition. Il est en principe possible de contourner le problème, mais cela peut mener à des obligations de preuve plus fortes, donc à un effort de preuve plus important. Par exemple, en notation *B classique*, on pourrait écrire ***pre*** $Go_i \Rightarrow Po_i$ ***then*** *(****if*** Go_i ***then*** *... **else** skip)* pour spécifier une opération de précondition Po_i gardée par Go_i. Néanmoins, l'obligation de *maintien de l'invariant* serait alors plus forte, en particulier dans le cas où on a $Goa_i(a) \wedge$

$\neg Goc_i(c)$, il faudrait montrer $I(a,c) \wedge Poa_i(a) \Rightarrow I(oa_i(a),c)$, alors que ce cas est trivialement vrai dans notre approche.

La démarche de raffinement est proche de l'interprétation abstraite. En particulier la condition de correction d'un interprète abstrait est une condition de raffinement vis-à-vis de l'invariant de liaison $I(a,c) \equiv c \in \gamma(a)$, où γ est une *fonction de concrétisation* [17]. Cependant, les définitions fonctionnelles sont contraintes dans le formalisme de *Coq* (notamment pour assurer la totalité). Lorsque ces contraintes sont trop fortes, il est possible de recourir aux prédicats inductifs qui sont plus expressifs, ou à une axiomatisation de γ [5], mais cela nécessite un plongement profond. En revanche, la définition d'un invariant de liaison, de nature essentiellement logique, est assez naturelle dans un formalisme basé sur la théorie des types.

5. Conclusion

Nous avons proposé une application de la démarche de raffinement à la preuve de correction d'une analyse de programmes. Une fois la correction prouvée, une implémentation exécutable de l'analyseur consiste à parcourir l'espace d'état abstrait en appliquant *AbstractExec* à l'état abstrait initial, puis à tous les états atteignables par application de cette fonction. A chaque fois qu'un nouvel état est généré, on s'assure que ce dernier est sûr en évaluant les préconditions des instructions qui peuvent être exécutées dans cet état. Si l'état n'est pas sûr la vérification échoue et l'algorithme s'arrête. Si l'algorithme termine sans rencontrer d'état ne satisfaisant pas les critères de sûreté, alors la vérification a réussi. A moins qu'il soit possible d'exhiber une mesure décroissante, la convergence de cet algorithme n'est pas garantie, par conséquent il faut se donner un nombre maximum d'itérations. L'aspect de la convergence reste donc à approfondir. Il serait également intéressant de pouvoir procéder à une extraction certifiée de code à partir de la sémantique de l'analyseur. Enfin, il est nécessaire de garder à l'esprit le risque d'explosion de l'espace d'état durant la conception de la sémantique abstraite.

L'approche présentée dans cet article est actuellement appliquée dans le cadre d'une étude de cas qui vise une analyse de bytecode capable de détecter des erreurs de programmation liées à l'*aliasing* [13] dans les programmes objet. On s'intéresse à l'échappement des références qui peut constituer une violation de l'encapsulation et ainsi aboutir à des problèmes d'intégrité de données. Pour éviter ces situations, on restreint le domaine de l'instruction *setfield* au cas où la référence cible est présente dans la pile d'appels. Actuellement, le développement [4] *Coq* compte environ 4200 lignes de script, et une centaine de preuves. Sur le plan pratique, la démarche de raffinement donne un cadre formel de réflexion. Le recours à un plongement léger dans un style opérationnel permet de simplifier les preuves. En particulier, cela permet de tirer avantage des tactiques de simplification, et les concepts manipulés sont assez proches de leurs descriptions formelles, ce qui aide à maîtriser la complexité.

4. Sources *Coq* sur `https://github.com/bsall/afadl-2017`

Références

[1] J.-R. Abrial. *Modeling in Event-B : system and software engineering*. Cambridge University Press, 2010.

[2] J.-R. Abrial and J.-R. Abrial. *The B-book : assigning programs to meanings*. Cambridge University Press, 2005.

[3] R. Back. Refinement calculus, part ii : Parallel and reactive programs. In *Stepwise Refinement of Distributed Systems Models, Formalisms, Correctness*, pages 67–93. Springer, 1990.

[4] R.-J. Back and J. Wright. *Refinement calculus : a systematic introduction*. Springer Science & Business Media, 2012.

[5] Y. Bertot. Structural Abstract Interpretation : A Formal Study Using Coq. In *Language Engineering and Rigorous Software Development*, pages 153–194. Springer, 2009.

[6] S. Boulmé. Intuitionistic refinement calculus. In *International Conference on Typed Lambda Calculi and Applications*, pages 54–69. Springer, 2007.

[7] C. Cohen, M. Dénes, and A. Mörtberg. Refinements for free ! In *International Conference on Certified Programs and Proofs*, pages 147–162. Springer, 2013.

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

[9] B. Delaware, C. Pit-Claudel, J. Gross, and A. Chlipala. Fiat : Deductive synthesis of abstract data types in a proof assistant. In *ACM SIGPLAN Notices*, volume 50, pages 689–700. ACM, 2015.

[10] P. Gardiner and C. Morgan. Data refinement of predicate transformers. *Theoretical Computer Science*, 87(1) :143–162, 1991.

[11] J. He, C. Hoare, and J. W. Sanders. Data refinement refined resume. In *European Symposium on Programming*, pages 187–196. Springer, 1986.

[12] C. A. R. Hoare. Proof of correctness of data representations. In *Software pioneers*, pages 385–396. Springer, 2002.

[13] J. Hogg, D. Lea, A. Wills, D. de Champeaux, and R. Holt. The Geneva Convention on the Treatment of Object Aliasing. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, pages 7–14. Springer, 2013.

[14] A. Joao and S. Wouter. Embedding the refinement calculus in coq. *SCP (submitted)*, 2016. `http://www.staff.science.uu.nl/~swier004/publications/2017-scp-draft.pdf`.

[15] R. Miarka, E. Boiten, and J. Derrick. Guards, preconditions, and refinement in Z. In *International Conference of B and Z Users*, pages 286–303. Springer, 2000.

[16] J. M. Spivey and J. Abrial. *The Z notation*. Prentice Hall Hemel Hempstead, 1992.

[17] A. Spiwack. Abstract interpretation as anti-refinement. *arXiv preprint arXiv :1310.4283*, 2013.

[18] C. D. Team et al. The Coq proof assistant reference manual Version 8.5. *TypiCal Project (formerly LogiCal)*, 2015.

[19] N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4) :221–227, 1971.