Analyse simple de types dans les tableaux et optimisation du ramasse-miettes.

Dominique Colnet
LORIA - Université de Lorraine
http://SmartEiffel.loria.fr

Benoît Sonntag
LSIIT - Université de Strasbourg
http://www.lisaac.org

Abstract

This article starts with the presentation of a simple technique, using type flow analysis and filling up order, to predict the *content* of arrays. Applied first on low-level arrays indexed from 0, our technique is then extended to deal with higher level data structures using arrays, like variable index arrays as well as circular arrays and hash-maps.

The main aim of our technique is to allow the propagation of the type flow information through array read-write expressions, thus opening the last gate to a global type flow analysis.

Beside the improvement of type prediction useful for dynamic binding and type security of object-oriented languages, our technique makes it possible to optimize memory management. Indeed, thanks to the filling up order, the garbage collector (GC) only inspects the used part of arrays, avoiding collection of unused objects referenced by the supply part of arrays. Furthermore, the supply part of arrays does not even require initialization or cleaning after use.

Measurements we present show the global improvement on type flow analysis and the real gain during the mark and sweep of arrays.

Résumé

Cet article commence en présentant une technique très simple, par analyse de flots de types et ordre de remplissage imposé, permettant de prédire le *contenu* des tableaux. D'abord présentée pour les tableaux primitifs indexés à partir de 0, notre technique est ensuite étendue pour prendre en compte les autres structures de données de plus haut niveau: tableaux à indexation variable, tableaux circulaires et tables de hachage.

Le résultat essentiel consiste à pouvoir faire suivre l'information de flots de types déjà collectée pour le reste du code source au travers des expressions qui manipulent des tableaux, permettant ainsi de procéder à une analyse de flots de types vraiment globale.

En plus de l'amélioration de la prédiction de types utile pour la liaison dynamique et la sécurité des langages à objets, notre technique permet d'optimiser la gestion mémoire. En effet, grâce à la technique de remplissage utilisée, le ramasse-miettes (GC) n'inspecte que les zones utiles des tableaux en évitant de collecter des objets inaccessibles, référencés par les zones de réserve. Ces zones de réserve n'ont par ailleurs nullement besoin d'être initialisées avant utilisation ni nettoyées aprés usage.

Les mesures présentées permettent de se rendre compte de l'impact de cette technique, aussi bien en terme de qualité de l'analyse de flots, qu'en terme de gain au niveau de la gestion mémoire durant le marquage et le balayage des tableaux.

1 Introduction et cadre du travail

Dans le contexte des langages à objets, la prédiction de types dynamiques permet d'optimiser l'implantation de la liaison dynamique ainsi que de détecter certains appels sur NULL [1, 2, 3, 4, 5, 6, 7, 8]. Ces nombreux travaux concernent la prédiction du type des variables, variables globales, variables locales, variables d'instances ou encore arguments de fonctions. En ce qui concerne la prédiction des types des éléments contenus *dans* les tableaux, nous n'avons pas trouvé de travaux publiés. En effet, les articles concernant les tableaux se focalisent essentiellement sur

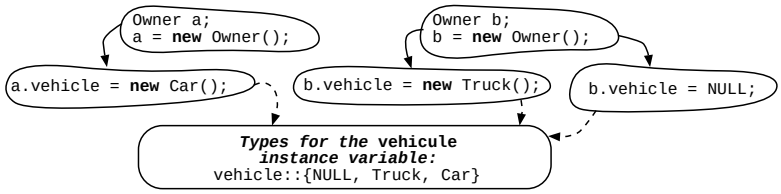

Figure 1: Analyse de flots de types pour la variable d'instance `vehicule`. L'ordre des instructions ne compte pas et le résultat est valable pour toutes les instances de la classe OWNER.

la vérification du respect des bornes [9, 10, 11]. L'analyse du contenu des tableaux est plus récente: [12, 13, 14, 15, 16, 17, 18, 19, 20] mais ne concerne pas l'analyse de flots de types.

Pour les langages à objets, il est important de réfléchir au polymorphisme à l'intérieur des tableaux. Cela arrive lorsque le tableau contient un mélange d'objets de types différents.

Pour aborder la prédiction de types du contenu des tableaux avec un résultat aussi précis que possible, nous procédons à une analyse globale de l'ensemble du code source. Après calcul de la fermeture transitive du graphe d'appel, nous disposons de l'ensemble du code vivant, plus exactement, un sur-ensemble du code qui sera réellement exécuté. En effet, pour les tableaux comme pour les autres variables, il s'agit de recenser tous les types servant en écriture afin de prédire le résultat des lectures. Considérer trop de code risque d'ajouter des types ne pouvant pas se trouver dans une véritable exécution, ce qui revient à rendre imprécise la prédiction de types. Inversement, il est impensable d'oublier du code vivant car cela risque d'enlever des types pouvant survenir à l'exécution. Dans la suite, quand nous utilisons l'expression *code*, il s'agit toujours du *code vivant*, c'est à dire le code qui a une chance d'être exécuté. C'est sur cette base que nous effectuons la prédiction de types.

1.1 Flots de données *vs* flots de types et valeurs NULL

Pour la prédiction de types, nous effectuons une *analyse de flots de types*, qu'il ne faut pas confondre avec une *analyse de flots de données*. L'analyse de flots de données consiste à tenir compte de l'ordre des instructions dans le but d'obtenir une information très précise, voir même exacte, c'est à dire permettant de savoir quel est véritablement l'objet référencé par une expression. L'analyse de flots de données permet d'affiner l'information de l'analyse de flots de types, mais n'est pas nécessaire pour notre technique de prédiction de types sur les éléments d'un tableau.

Ce que nous appelons analyse de flots de types consiste à calculer pour une expression donnée, l'ensemble des types dynamiques possible *sans tenir compte de l'ordre des instructions*. Par exemple, grâce à l'ensemble des affectations possibles d'une variable, on calcule l'union des types possibles pour cette variable. L'ordre des affectations à cette variable ne compte pas essentiellement pour des raisons de passage à l'échelle. On considère globalement l'ensemble des affectations qui sont dans le code. En ce qui concerne les variables d'instances, on ne distingue pas une instance d'une autre (voir figure 1); l'information concerne globalement toutes les instances de la classe. Pour un argument de méthode, on utilise l'ensemble de tous les arguments effectifs dans tous les appels du code. Pour obtenir l'ensemble des types dynamiques possibles pour un appel de méthode on propage les informations concernant le receveur et les arguments.

Il va sans dire qu'une analyse de flots de types est nettement moins coûteuse en temps de calcul qu'une analyse de flots de données. L'objectif de cet article consiste à détailler la

propagation de l'information de types à travers les lectures/écritures dans les tableaux.

Une variable non initialisée est une source bien connue d'erreurs : entiers avec des valeurs aléatoires et imprédictibles, pointeurs en dehors des zones autorisées, etc. En outre, une variable non initialisée introduit une incertitude qui rend pratiquement inutile l'analyse de flots de types. Laisser la possibilité d'utiliser une variable non initialisée revient à introduire la possibilité d'avoir des valeurs aléatoires dans l'ensemble des types possibles d'une variable. Le langage doit donc, soit forcer le programmeur à initialiser les variables (e.g. locales de Java), soit fixer une valeur par défaut aux variables non initialisées (e.g. Eiffel). L'impossibilité d'avoir une variable non initialisée est devenue la règle dans les langages de programmation récents.

Dans la suite, la valeur NULL, qui représente l'absence de référence est simplement considérée comme un type particulier possible et sa présence dans l'ensemble des types associés à une expression indique que cette dernière peut prendre la valeur NULL. Notons que dans le cas d'un langage comme Java, sans valeur par défaut pour les variables, la présence de NULL implique qu'il existe dans le code une affectation explicite avec NULL ou, par transitivité, une affectation avec une autre expression pouvant prendre cette valeur. Par exemple, dans la figure 1, l'ensemble des types possibles de la variable `vehicule` contient NULL.

1.2 Utilité de l'analyse de flots de types

Il est possible que le résultat de l'analyse de flots de types pour une variable soit réduit à l'ensemble {NULL}. Si cette variable est la cible d'un appel de méthode dans le code vivant, il s'agit d'un appel invalide détecté statiquement. Inversement, si l'ensemble des types possibles ne contient pas NULL, l'appel sera toujours valide à l'exécution, et détecté statiquement comme tel. En outre, dans certains cas, l'ensemble des types possibles d'une expression est réduit à un seul type non NULL. Un tel cas de figure permet un appel statique, sans liaison dynamique, voire même, l'inlining de la méthode appelée.

Dans le cas d'un langage typé statiquement, l'information collectée par analyse de flots de types vient s'ajouter à l'information du type statique. Tous les types dynamiques de l'ensemble sont des sous-types du type statique. Dans la pratique, cette information est souvent même plus précise, transformant le type statique en information pour le programmeur et en simple contrôle de cohérence pour le compilateur. Dans le cas d'un langage non typé statiquement, l'ensemble de types dynamiques est la seule information disponible. Celle-ci est donc encore plus importante et pourrait même servir de documentation automatique pour le programmeur. Obtenir de l'information de type pour le contenu des tableaux permet de prendre en compte toutes les sortes d'expressions, permettant ainsi, par transitivité, d'avoir une information de type sur la totalité d'un code source ne comportant aucune indication statique de type.

Pour l'implantation de la liaison dynamique avec des VFTs (*Virtual Function Tables*) [21, 22], l'ensemble de types dynamiques possibles pour le receveur étant connu, cela permet de figer et de minimiser la taille des tables. Il est également possible d'implanter la liaison dynamique par du code de sélection dans l'ensemble des types dynamiques possibles, soit avec un `switch`, soit par sélection dichotomique [7].

2 Analyse de flots de types dans les tableaux

En Java, un tableau élémentaire possède une information donnant sa taille, l'attribut `length`. On souhaite également pouvoir adapter notre technique aux tableaux les plus élémentaires comme ceux que l'on trouve par exemple en C, composés d'une simple zone de mémoire consécutive, sans indication de capacité.

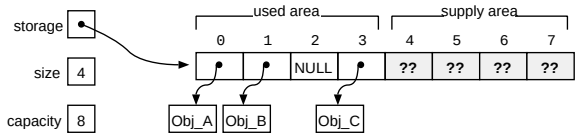

Figure 2: Manipulation d'un tableau avec trois variables: `storage`, `size` and `capacity`.

2.1 Remplissage de la gauche vers la droite

Le principe de base consiste à faire en sorte que le tableau se remplisse progressivement de la gauche vers la droite, sans jamais laisser de places non initialisées. L'utilisateur doit bien entendu garder la possibilité d'utiliser NULL comme une valeur à part entière, par exemple dans le cas de l'utilisation d'un tableau comme table de hachage. Ceci empêche donc d'utiliser NULL comme marqueur de fin de zone utile.

La manipulation d'un tableau peut être vue comme la manipulation d'un type de données abstrait regroupant trois variables (figure 2). La variable `storage` sert à pointer la zone mémoire de stockage. La variable `size` indique la taille de la zone de gauche du tableau qui correspond à la partie déjà utilisée. La partie droite correspond à la zone du tableau non encore utilisée, la zone en réserve. Pour finir, la variable `capacity` indique la taille totale de la zone de stockage, la zone potentiellement utilisable. Les opérations du type abstrait pour manipuler un tableau sont les suivantes:

(C) Création du tableau Lors de la création d'un tableau, les trois instructions qui suivent doivent être dans la même séquence:

```
capacity = some_positive_integer_expression;
size = 0;  storage = malloc(capacity);
```

(R) Lecture dans la zone déjà utilisée Pour la lecture, il faut que *some_index* soit compris entre `0` et `size - 1`:

```
some_variable = storage[some_index];
```

(W) Ecriture dans la zone déjà utilisée De même, pour l'écriture, il faut que *some_index* soit compris entre `0` et `size - 1`:

```
storage[some_index] = expression;
```

(A) Ajout d'un élément dans la zone de réserve S'il reste de la place dans la zone de réserve (`size` < `capacity`), il est possible d'ajouter un élément avec:

```
storage[size] = expression;
size = size + 1;
```

(F) Libération du tableau Pour libérer le tableau après utilisation:

```
capacity = 0;
size = 0;
free(storage);
storage = NULL;
```

Pour que l'utilisation du tableau soit correcte, les séquences *(C)*, *(R)*, *(W)*, *(A)* et *(F)* doivent être utilisées dans un ordre respectant l'automate de la figure 3. Cet automate modélise le fait qu'il faut commencer par la création du tableau *(C)* avant de faire une première adjonction *(A)*, puis, sans préciser un ordre particulier, des lectures *(R)*, des écritures *(W)*, de nouvelles adjonctions *(A)*, pour finir enfin par la libération du tableau *(F)*.

Tout utilisation ne respectant pas l'ordonnancement de l'automate est une utilisation erronée du tableau. Néanmoins, le respect de cet automate ne garantit pas que les indices d'accès dans le tableau sont valides. L'indice d'accès peut, soit sortir complètement de la zone mémoire, soit effectuer un accès dans la zone de réserve, la zone non initialisée. Il est bien entendu possible

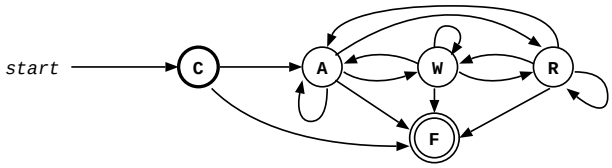

Figure 3: Utilisation correcte d'un tableau schématisé sous la forme d'un automate.

d'équiper les séquences *(R) (W)* et *(A)* avec du code de vérification de sortie de bornes. C'est la solution qui est actuellement utilisée dans la bibliothèque de SmartEiffel et de Lisaac, à l'aide de préconditions activables ou non lors de la compilation.

2.2 Information de flots de types

Pour la prédiction du type dynamique des éléments du tableau, nous faisons abstraction complète de l'endroit d'écriture ou de lecture. Tout se passe comme si le tableau était réduit à une seule et unique case. Par exemple, si un objet de type TRUCK est écrit à l'index 1 du tableau, on considère que tous les autres index du tableau peuvent potentiellement renvoyer un objet de type TRUCK. Si une autre séquence d'écriture sur le même tableau est détectée avec par exemple le type CAR, on considère alors que tous les accès à ce tableau peuvent retourner soit le type TRUCK, soit le type CAR.

Plus généralement, si S_a est l'ensemble des types dynamiques possibles pour le contenu du tableau avant l'écriture d'une expression ayant S_e comme ensemble de types dynamiques, alors S_a est recalculé par $S_a \cup S_e$.

Dans l'implantation du compilateur, pour mémoriser l'information de types d'un tableau il est possible de procéder de deux manières. La première consiste à tenir à jour l'ensemble des types pour le tableau, par unions successives, avec les ensembles de types des expressions écrites dans le tableau. On applique alors simplement la définition donnée précédemment. Si les ensembles de types pour les expressions écrites dans le tableau ne sont pas disponibles, ou incomplètes, en particulier au début du processus de compilation, il est possible de procéder autrement. La deuxième solution consiste à dresser, pour le tableau concerné, la liste des expressions écrites dans ce tableau, par simple adjonction. L'ensemble de types du tableau peut ensuite être calculé à la demande, par propagation, lorsque chacune des expressions mémorisée dans la liste possède elle-même un ensemble de types connus (i.e. fermeture transitive du graphe).

2.3 Contrôle du respect de l'utilisation des séquences

Le compilateur doit vérifier que les séquences *(C)*, *(R)*, *(W)*, *(A)* et *(F)* sont bien présentes et si possible, bien ordonnées dans le code par rapport à l'automate. Comme nous le montrons dans la suite, la présence des séquences est facilement vérifiable. En revanche, montrer que l'ordonnancement des séquences est correct par rapport à l'automate revient à essayer de mettre en correspondance le graphe d'appel du code avec le graphe de l'automate. Ce calcul est extrêmement difficile et, sauf pour quelques cas simples, il ne faut pas espérer pouvoir le faire statiquement de manière générale.

Certaines erreurs élémentaires sont néanmoins assez simples à détecter, comme l'absence

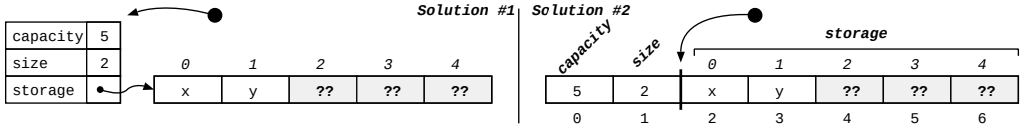

Figure 4: Deux solutions pour unifier l'accès aux variables `capacity`, `size` et `storage`.

dans le code source de la séquence *(C)* qui indique la non initialisation du tableau. L'absence de *(F)* indique l'oubli de la libération mémoire du tableau. Notons également que l'absence totale de séquence d'écriture, *(A)* ou *(W)* est suspecte. Si en plus il n'y a pas de lecture *(R)*, il s'agit probablement d'un tableau vide non utilisé. S'il y a des lectures il s'agit d'une erreur pouvant être signalée statiquement.

Dans le cas de présence d'écritures *(W)* et d'absence de lecture *(R)*, comme le tableau n'est pas utilisé, il est soit possible d'avertir le programmeur, soit d'effacer complètement de l'exécutable le code de toutes les autres séquences.

S'il est difficile de vérifier la concordance du code source avec l'automate, il est cependant possible de vérifier la présence des séquences dans le code source vivant. Une première solution consiste pour le compilateur à contrôler dans la représentation abstraite du code source la présence des séquences *(C)*, *(R)*, *(W)*, *(A)* et *(F)*. Cette détection pratiquement syntaxique est contraignante pour le programmeur, celui-ci ne pouvant même pas, par exemple, permuter les instructions de la séquence *(F)*. On peut imaginer une forme de tolérance de la part du compilateur, en autorisant par exemple, les quelques permutations qui ne violent pas les propriétés de remplissage et d'initialisation du tableau. Il est plus difficile de localiser une séquence quand elle est diffusée dans du code, sachant qu'il faut vérifier également que ce code entremêlé est sans effet de bords sur la séquence elle même.

Afin de remédier au problème de détection des séquences et pour permettre une utilisation simple de ces séquences par le programmeur, il suffit d'encapsuler les composants du tableau, `storage`, `size` et `capacity` comme présenté dans la figure 4 et d'introduire des *builtins* de manipulation. La solution 1, adaptée aux langages à objets, isole les variables d'accès au tableau dans une structure, avec un pointeur unique à disposition du programmeur. Le builtin `new(initial_capacity)` contient le code de la séquence `(C)` et retourne le pointeur sur le tableau. Le builtin `read(array, index)` contient le code de la séquence `(R)` et retourne la valeur lue. Le builtin `write(array, index, value)` contient le code de la séquence `(W)` et écrit dans le tableau; ainsi de suite pour les séquences `(A)` et `(F)`. Comme la solution 1 présente l'inconvénient d'une indirection supplémentaire pour accéder au `storage`, il est possible d'utiliser la solution 2 de la figure 4.

Finalement, même si on ne peut pas facilement vérifier la cohérence de l'automate figure 3 avec le code, l'essentiel est de pouvoir mettre en œuvre l'analyse de flots de types dans les tableaux.

2.4 Optimisation avec des builtins supplémentaires

Nous avons implanté et expérimenté la technique d'analyse de flots de types dans les tableaux pour notre compilateur Lisaac [23]. A l'usage, nous avons constaté qu'il était plus pratique d'introduire des séquences de manipulation supplémentaires pour les tableaux. De même dans la bibliothèque de SmartEiffel [24], une technique de remplissage gauche-droite est utilisée, ainsi que ces builtins supplémentaires.

Sauf le builtin `size_reduction` présenté ci-dessous, tous les nouveaux builtins peuvent être définis en utilisant les séquences de base. Le builtin `size_reduction` ne viole pas les principes essentiels de manipulation des tableaux.

Nous avons introduit le builtin `calloc`, pour allouer et initialiser d'un seul coup un tableau. Ce nouveau builtin équivaut à la séquence suivante:

```
array* calloc (initial_capacity) {
  result = new(initial_capacity); // Sequence (C)
  for (i = 0 ; i < initial_capacity ; i++) {
    add_last(result, NULL); // Sequence (A)
  };
  return result;
};
```

Nous avons eu besoin de ce builtin pour optimiser l'implantation de toutes les collections à base de hachage dans la bibliothèque de Lisaac (dictionnary, set, ...). L'implantation d'une table de hachage nécessite la création d'un tableau avec une zone utile initialisée à NULL et occupant l'ensemble de sa capacité. Notons que pour des raisons de performances, l'implantation de `calloc` peut utiliser un `memset` du langage C afin de profiter des instructions spécifiques du processeur pour initialiser des zones de mémoire.

Le builtin `size_reduction` à utiliser pour réduire la taille de la zone utile d'un tableau est le suivant :

```
size_reduction (array, new_size) {
  if ((0 <= new_size) && (new_size <= array.size))
    array.size = new_size;
  } else {
    //  Runtime exception
  };
};
```

Ce builtin est nécessaire pour utiliser un tableau comme un buffer temporaire, qui grossit et rétrécit au gré de l'exécution. Il ne permet que de réduire la taille de la zone utile et ne change donc pas l'information de flots de types. Comme nous le verrons également dans la suite, la possibilité de réduire la zone utile facilite le travail du GC.

Enfin, nous avons également ajouté le builtin `reallocation`, à utiliser pour augmenter la capacité d'un tableau existant:

```
array* reallocation (array, new_capacity) {
  result = new(new_capacity); // Sequence (C)
  for (j = 0 ; j < array.size ; j++) {
    add_last(result,read(array,j)); // Sequence (A)
  };
  free(array); // Sequence (F)
  return result;
};
```

Ce nouveau builtin, uniquement décrit avec les séquences existantes, ne change pas non plus l'information de flots de types. Notons qu'il est possible d'utiliser la fonction `realloc` du langage C pour optimiser l'implantation et éventuellement éviter les déplacements en mémoire.

3 Ramasse-miettes et collections

Avant de présenter l'utilisation du remplissage gauche droite sur l'implantation des principales collections de haut niveau, étudions l'impact de l'information de flots de types sur le ramasse-miettes (GC).

3.1 Ramasse-miettes dans les tableaux

La plupart des algorithmes de GC [25], comme le *mark-and-sweep* ou le *copying-collector*, ont besoin de parcourir l'ensemble des objets accessibles. Par exemple, lors de la phase de marquage, l'algorithme *mark-and-sweep* doit visiter tous les tableaux accessibles. Pour chaque tableau accessible, le GC doit ensuite parcourir toutes les cases de ce tableau afin de continuer le marquage.

La zone de réserve des tableaux (figure 2) est une zone de mémoire non initialisée. Quand le GC parcoure cette zone, il risque de rencontrer, la plupart du temps, des pointeurs invalides ou, aléatoirement, des références vers des objets qui ne sont pas forcément accessibles par un autre chemin. Inversement, la zone utile du tableau est toujours correctement initialisée et ne contient que des références valides incluant éventuellement des NULLs. Il est possible pour le GC d'utiliser l'indice `size` afin d'éviter de parcourir inutilement la zone de réserve. Ceci permet d'une part d'aller plus vite, mais aussi d'éviter de marquer des objets non accessibles par le programme.

En outre, l'information de flots de types du tableau peut également être intégrée au GC comme cela se fait déjà pour les attributs des objets [26]. En effet, SmartEiffel génère une fonction de marquage spécialisée, conservative [25], pour chaque type d'objet, sans interprétation lors de l'exécution. Généré différemment pour chaque application, le code du GC adapté aux objets manipulés par cette application, est intégré dans l'exécutable. Le code de marquage intègre la connaissance de la structure des objets. Un attribut de type simple comme par exemple un entier, un caractère ou un nombre flottant est tout simplement ignoré (pas de génération de code de marquage) car il ne constitue pas un chemin possible vers un autre objet. Seuls les attributs qui sont des références sont inspectés. En outre, selon que l'attribut est monomorphe ou polymorphe, l'appel de la fonction de marquage correspondante se fait par un appel direct ou par un code similaire à celui de la résolution de la liaison dynamique.

La méthode utilisée pour les attributs est applicable pour les tableaux, par itération dans la zone utile. Dans le cas où les références utiles d'un tableau ne sont jamais NULLs, on peut économiser un test pour chaque case d'un tableau inspectée. Si tous les éléments d'un tableau ont par exemple un même et unique type dynamique possible, on évite l'équivalent d'une liaison dynamique pour chaque case du tableau.

3.2 La classe FAST_ARRAY

La classe FAST_ARRAY de la bibliothèque de SmartEiffel correspond à l'application directe de la technique de remplissage progressive de la gauche vers la droite avec l'utilisation de la solution 1 de la figure 4. Afin d'éviter une indirection supplémentaire, les trois variables de manipulation du tableau, `storage`, `size` et `capacity`, sont directement les attributs de la classe FAST_ARRAY. Le GC de SmartEiffel, produit de façon adaptée pour chaque application [26] utilise également l'information `size` afin de ne pas parcourir, durant la phase de marquage, la zone de réserve du tableau.

Le builtin `size_reduction` présenté précédemment, possède également son équivalent dans la bibliothèque de SmartEiffel et permet ainsi d'informer dynamiquement le GC des réductions

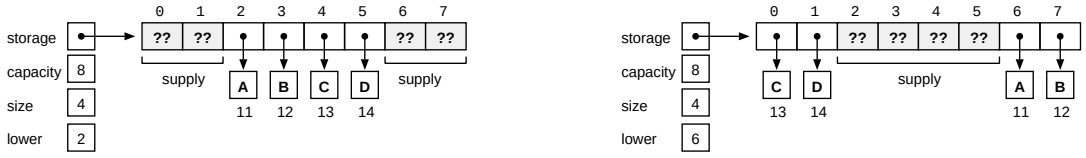

Figure 5: Tableaux circulaires. Deux rangements possibles pour les mêmes données.

de la taille de la zone utile de chaque tableau. En cas de réduction de `size`, la zone de réserve ne contient plus de cases ayant des valeurs aléatoires, non initialisées, mais seulement des NULLs ou des références valides vers des objets existants. Il est d'autant plus important de transmettre cette information au GC afin d'éviter le marquage des objets correspondants.

Le builtin `reallocation` est utilisé quand un objet de la classe FAST_ARRAY doit être redimensionné, en changeant sa `capacity`. L'attribut `size` n'étant pas affecté, il n'y a pas de conséquence pour le GC ni sur l'information de flots de types.

Notons également que le compilateur, s'il n'implante pas totalement la vérification syntaxique des séquences de modifications, force l'utilisation de noms figés pour les trois attributs, `storage`, `size` et `capacity`, de manipulation du tableau. La vérification stricte des séquences pourrait être facilement ajoutée sans être gênante pour l'utilisateur, car localisée et donc écrite une seule fois dans la classe FAST_ARRAY. En fait, le compilateur utilise également l'information `size` pour générer des fonctions de comparaison/copie en profondeur, plus efficaces car ne considérant pas la zone de réserve.

3.3 La classe ARRAY

Un objet de la classe ARRAY représente un tableau dont l'index de départ est quelconque, positif ou négatif et non pas figé à 0 comme un objet de la classe FAST_ARRAY. La technique de remplissage de la gauche vers la droite est trivialement applicable dans ce cas de figure, modulo les translations d'index habituelles. Seul l'ajout d'une variable servant de sauvegarde de la translation en vigueur est nécessaire dans la structure d'un objet ARRAY. Cette variable n'entre pas directement en jeu dans l'utilisation de nos séquences.

3.4 La classe RING_ARRAY

Nous appliquons une technique similaire dans le cas des tableaux circulaires en enrichissant le type abstrait de manipulation avec la variable `lower` (voir figure 5). Les opérations de manipulation du type abstrait ont été revues en conséquence sans poser de problème particulier.

3.5 Les structures de données à base de hachage

Pour compléter la bibliothèque avec les structures de données à base de hachage, comme par exemple pour les ensembles, on utilise habituellement une table primaire pouvant contenir des NULLs ou des cellules de stockage (figure 6). L'accès dans la table primaire se fait grâce au *hash code* de l'objet recherché. La case correspondante contenant un NULL indique l'absence de l'objet recherché dans l'ensemble. La case correspondante aboutissant sur une cellule de stockage permet de continuer la recherche en tenant compte des éventuelles collisions.

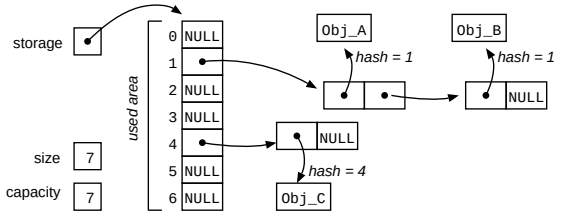

Figure 6: Représentation d'un ensemble par méthode de hachage. La zone de réserve est vide.

Pour créer la table primaire de hachage, on utilise le builtin `calloc`, puis, pour lire et écrire, les séquences `(R)` et `(W)`. Il n'est pas nécessaire d'introduire de nouvelle séquence. La table primaire ne comportant pas de zone de réserve, le GC parcoure normalement l'ensemble de la table primaire. Comme il est normal que la table primaire puisse contenir des NULLs, l'utilisation de `calloc` est justifiée et implique très normalement la présence de NULL dans l'ensemble de types de la table primaire.

4 Résultats expérimentaux

4.1 Mesures dans la bibliothèque de SmartEiffel

Nous avons instrumenté le code du GC de SmartEiffel afin d'inspecter l'impact de la gestion des tableaux sur le comportement mémoire.

Afin d'avoir une trace d'exécution réaliste, nous utilisons le code source du compilateur SmartEiffel lui même, soit 180,000 lignes de code Eiffel, durant son bootstrap, pour effectuer toutes les mesures qui suivent. La recompilation du compilateur par lui-même constitue un excellent banc d'essai car il utilise un grand nombre de tableaux et nécessite environ 330 Mo de mémoire durant son exécution. En outre, le GC est déclenché 32 fois lors de chaque recompilation.

Toutes les mesures qui suivent concernent exclusivement les tableaux de références vers des objets. Le contenu des autres tableaux, comme par exemple les tableaux d'entiers ou les tableaux de nombres flottants, ne sont pas examinés lors de la phase de marquage du GC.

Nous avons instrumenté la procédure de marquage des tableaux afin de comptabiliser le nombre de fois où cette routine est appelée durant une recompilation. Le GC procède à 6,399,198 marquages de tableaux. En cumulant les tailles des zones utiles parcourues durant ce marquage, on arrive au total de 12,548,963 cases de tableaux inspectés durant le marquage. On dispose aussi du cumul des `capacity` qui est de 21,714,957 cases. Grâce à l'optimisation consistant à ne pas marquer la zone de réserve, on constate que le GC a fait l'économie de 9,165,994 de cases, soit un gain de 42%. Les chiffres très précis que nous venons de donner sont les résultats d'une exécution en particulier car nous nous sommes apperçus d'une assez légère variation d'une exécution à l'autre. Parfois même, le nombre de fois où le GC se déclenche peut varier de 31 à 33. L'explication de cette variation réside dans le fait que le GC de SmartEiffel n'est pas conservatif [25, 26]: il considère toutes les valeurs de la pile comme étant des pointeurs potentiels vers des objets du programme. Selon les valeurs trouvées dans la pile à l'instant du marquage, le nombre d'objets supposés accessibles par le programme utilisateur peut donc changer, expliquant ainsi la variation constatée. Sur le pourcentage global de 42% de cases économisées, cette variation reste très légère, de l'ordre de 10^{-2}. N'importe quelle exécution

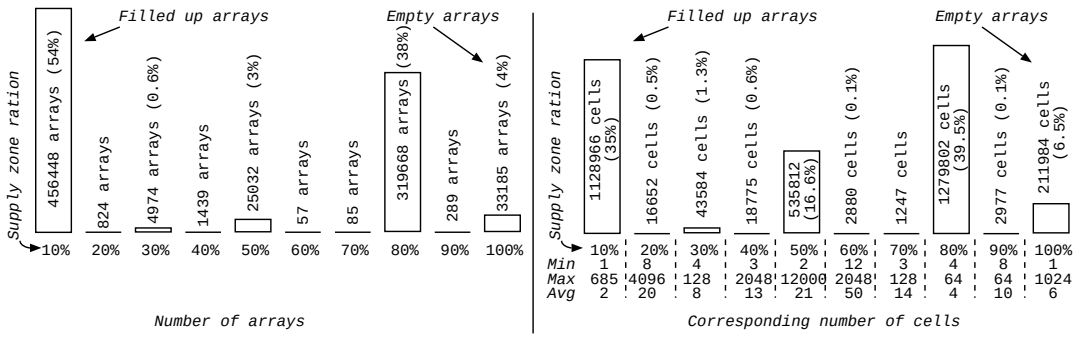

Figure 7: Mesure effectuée durant le dernier appel du GC lors du bootstrap de SmartEiffel.

du GC est donc significative et l'économie bien réelle.

Afin de mieux comprendre cet excellent résultat, nous avons examiné plus en détails la répartition des tableaux lors du dernier passage du GC à la fin de l'exécution du programme, juste avant l'`exit` final. Lors du dernier appel du GC, 838,681 tableaux sont inspectés en vu du marquage et parmi eux, 31,859 sont des tableaux vides. Les tableaux vides sont ceux où l'on tire le meilleur profit de l'optimisation. Il y a 453,215 tableaux complètement pleins. La partie gauche de la figure 7 donne la répartition des tableaux selon la taille relative de la zone de réserve. Les mesures concernant le dernier passage du GC sont encore plus stables que précédemment, de l'ordre de 10^{-3}.

Toujours durant le dernier passage du GC, nous avons mesuré que la taille totale des tableaux correspond à une capacité de stockage de 3,242,679 de références vers des objets dont 1,440,568 sont dans la zone de réserve (partie droite de la figure 7). Les valeurs *Min*, *Max* et *Avg* donnent respectivement pour chaque tranche de famille (famille 10%, famille 20%, etc.), la plus grande taille rencontrée, la plus petite taille rencontrée et la moyenne des tailles des tableaux de cette tranche. En moyenne, les tableaux sont donc très petits et les tableaux remplis occupent une grande place de la mémoire.

4.2 Mesures sur le compilateur Lisaac

Le compilateur Lisaac réalise une analyse de flots de types telle qu'elle vient d'être présentée dans cet article pour ce qui concerne le contenu des tableaux. Pour les mesures qui suivent, nous avons utilisé le code source du compilateur Lisaac lui-même durant son bootstrap, soit 53,000 lignes de code Lisaac.

Pour effectuer les mesures, nous avons modifié le code source du compilateur afin de connaître le niveau de variabilité des types à l'intérieur des tableaux (niveau du polymorphisme dans les tableaux). Bien entendu, seuls les tableaux de références sont intéressants du point de vue de l'analyse de flots de types. En outre, eux seuls peuvent contenir la valeur NULL. Comme expliqué précédemment, la valeur NULL est considérée comme un type particulier et est donc une valeur possible dans un ensemble de types dynamiques. La figure 8 donne la distribution des ensembles de types concernant l'intérieur des tableaux.

Le niveau de polymorphisme observé pour le code source du compilateur varie de 1 à 56, sachant qu'entre ces deux valeurs, tous les niveaux de polymorphisme ne sont pas rencontrés. La partie gauche de la figure 8 donne le nombre de tableaux pour un niveau de polymorphisme donné. Par exemple, il y a 129 tableaux ayant le niveau de polymorphisme 2 et un seul tableau

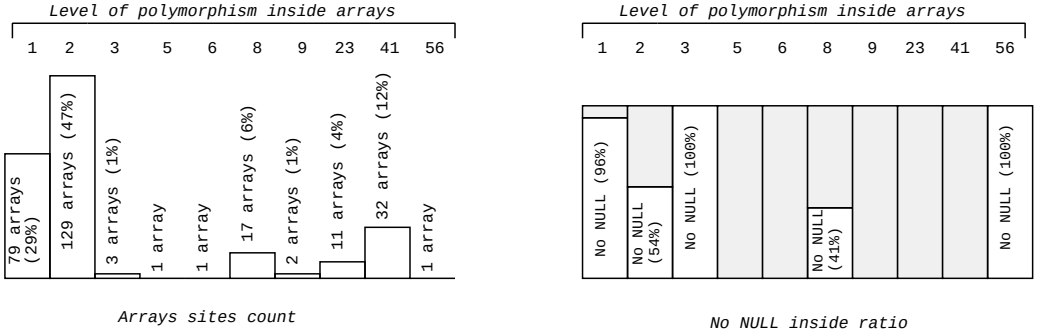

Figure 8: Distribution du polymorphisme dans les tableaux du code source de Lisaac.

avec le niveau de polymorphisme 56. On ne rencontre aucun tableau pour les niveaux 4, 7, 10, etc. Notons ici que le nombre de tableaux correspond à un nombre d'emplacements dans le code source et non pas aux nombre de tableaux que l'on rencontrera dynamiquement.

La partie droite de la figure 8 donne le ratio de type NULL en reprenant la même répartition par niveau polymorphique. On constate ainsi que tous les tableaux ayant les niveaux de polymorphisme 3 et 56 ne contiennent jamais le type NULL. Par ailleurs, si l'on totalise le nombre d'emplacements de tableaux ne contenant jamais NULL, on constate que 56.2% des tableaux ne comportent jamais la valeur NULL! Pour toutes les expressions de lecture concernant ces tableaux, le compilateur propage l'information et calcule statiquement les comparaisons avec NULL. Toujours sur la figure 8, la barre la plus à gauche indique que 79 emplacements de tableaux ne comportent qu'un seul et unique type. Ici encore, le compilateur propage cette information pour résoudre statiquement la liaison dynamique.

Pour observer la propagation de l'information de flots de types provenant des tableaux vers les autres variables, nous avons ensuite modifié le compilateur en introduisant artificiellement le type NULL dans les ensembles de types de tous les tableaux. Avant la modification, pour les 10,429 variables locales du code source du compilateur, on dénombre 3,202 variables, soit 30.7% incluant le type NULL dans leur ensemble de types dynamiques. Après modification, 7,043 variables locales sont infectées par le type NULL, soit 98.2% des variables locales. Cette expérimentation confirme donc le fait qu'une analyse de flots sur le contenu des tableaux améliore grandement les résultats d'analyse de flots sur le reste du code source.

La figure 9 donne le détail de la propagation de NULL pour les variables locales triées selon leur degré de polymorphisme qui varie de 2 à 57. La courbe de la partie basse reflète les valeurs que l'on trouve dans les bâtons de la partie haute. Notons ici que l'injection de NULL ne provoque pas une simple translation d'un cran vers la droite. Par exemple, pour les variables locales ayant le degré de polymorphisme 2 (la barre en haut à gauche), on dénombre 2046 variables avant injection et 5401 après injection. L'explication réside dans le fait que l'injection du type NULL dans un ensemble empêche de calculer statiquement les comparaisons avec NULL, rendant ainsi accessible le code préalablement mort.

La figure 10 présente la propagation de NULL des tableaux vers les variables globales et les variables d'instances. Pour les variables globales, on passe de 70.5% à 98.5%. Les variables d'instances n'échappent pas à la contamination en passant de 93.2% à 99.4%. Ce résultat montre une fois de plus l'intérêt de mettre en œuvre une analyse de flots de types dans les tableaux.

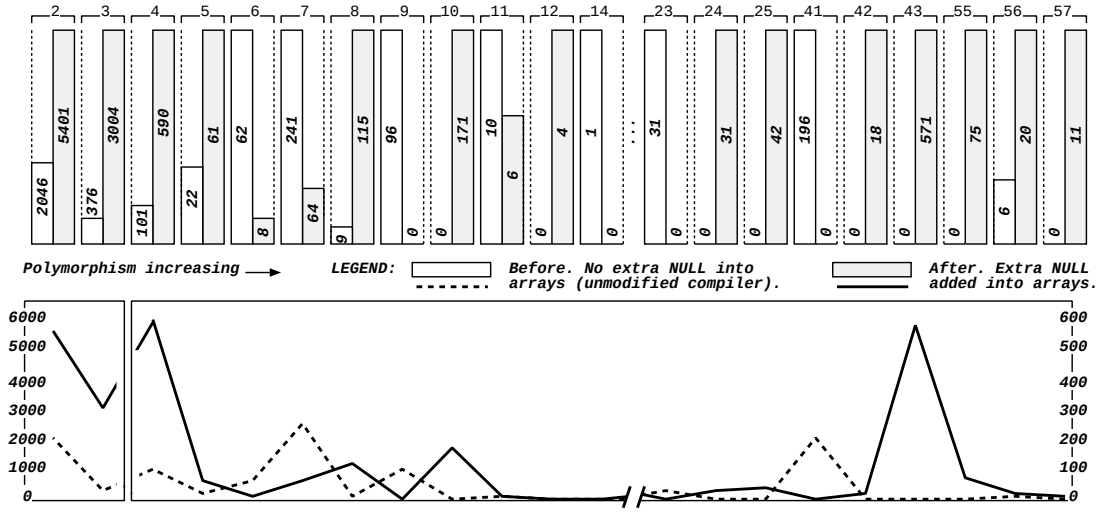

Figure 9: Propagation du type NULL des tableaux vers les variables locales. Mesures avant et après infection artificielle.

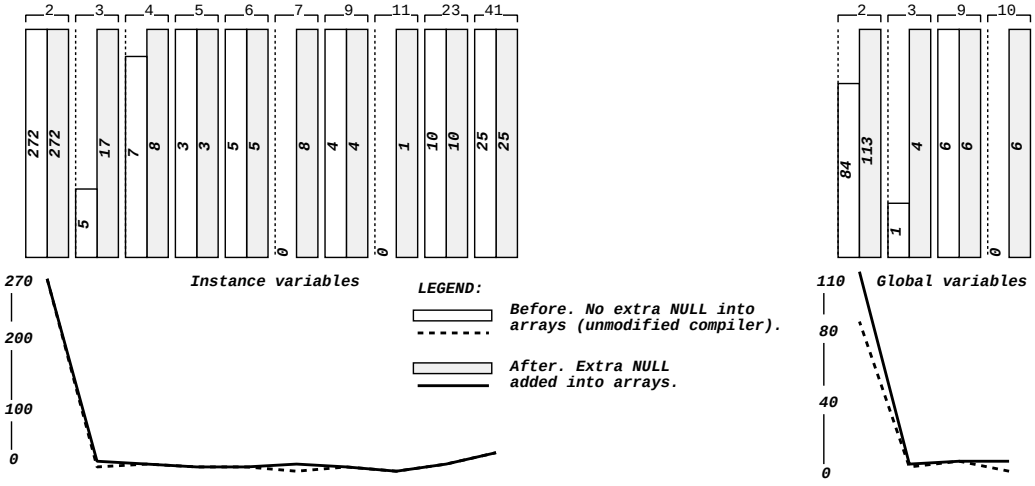

Figure 10: Propagation du type NULL vers les variables d'instance et vers les variables globales.

Pour poursuivre l'étude, nous avons artificiellement introduit non seulement NULL mais aussi tous les types dynamiques valides dans les tableaux. Par exemple, pour un tableau de VEHICLEs, tous les sous-types vivants et valides sont introduits artificiellement: CAR est ajouté, TRUCK aussi, BIKE également, etc., sans oublier NULL. Seul le résultat pour les variables locales est présenté sur la figure 11. Partant de 17 distributions de types avant modification, on passe, après modification à seulement 12 distributions, c'est à dire, sans surprise, une analyse nettement moins précise et donc nettement moins performante.

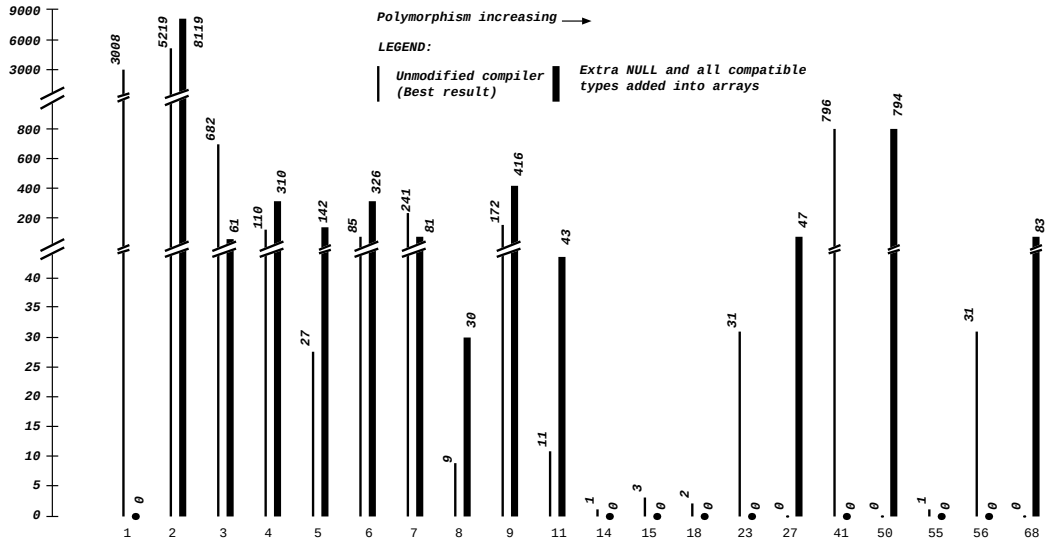

Figure 11: Propagation dans les variables locale après saturation artificielle des ensembles de types des tableaux.

5 Conclusion

La méthode présentée dans cet article force le remplissage progressif des tableaux permettant ainsi d'éviter la présence de cellules non initialisées à l'intérieur des tableaux sans avoir recours à une initialisation par défaut (i.e. `calloc`). En considérant globalement toutes les cases du tableaux sans tenir compte de l'indexation il est possible de collecter *à moindre frais* une information de flots de types à l'intérieur même des tableaux. Nous montrons par des mesures sur deux gros logiciels que cette information permet d'améliorer grandement la précision d'analyse de flots de types de manière globale pour toutes les expressions du langage. En outre, la technique que nous présentons permet d'améliorer sensiblement les performances du ramasse-miettes en évitant le balayage des zones de réserves des tableaux.

Nous montrons également dans cet article que la méthode de remplissage progressif n'est finalement pas si contraignante que cela. C'est cette technique qui est d'ailleurs utilisée depuis très longtemps dans les bibliothèques de SmartEiffel et de Lisaac pour toutes les collections de haut niveau, même par exemple pour les tableaux circulaires non présentés dans cet article.

Ce résultat obtenu sur les tableaux nous a également convaincu de privilégier l'initialisation explicite par le programmeur pour toutes les variables et non pas uniquement sur les tableaux. Par exemple, le choix effectué par les concepteurs du langage Java concernant l'initialisation des variables locales semble être le plus pertinent et se devrait d'être généralisé à toutes les catégories de variables ainsi que pour les tableaux bien sûr.

References

[1] Palsberg J, Schwartzbach MI. *Object-Oriented Type Inference.* OOPSLA'91, 146–161.

[2] Corney D, Gough J. *Type Test Elimination using Typeflow Analysis.* Proceedings of Programming Languages and System Architectures vol 792, LNCS, 1994, 137–150.

[3] Dean J, Grove D, Chambers C. *Optimization of object-oriented programs using static class hierarchy analysis.* ECOOP'95.

[4] Agesen O, Palsberg J, Schwartzbach MI. *Type Inference of Self: Analysis of objects with Dynamic and Multiple Inheritance.* Software Practice & Experience, 25(9), 1995, 975–995.

[5] Bacon DF, Sweeney PF. *Fast Static Analysis of C++ Virtual Function Calls.* OOPSLA'96, ACM Press, 1996, 324–341.

[6] Collin S, Colnet D, Zendra O. *Type Inference for Late Binding. The SmallEiffel Compiler.* JMLC'97, vol 1204, LNCS, 67–81.

[7] Zendra O, Colnet D, Collin S. *Efficient Dynamic Dispatch without Virtual Function Tables. The SmallEiffel Compiler.* OOPSLA'97.

[8] Fähndrich M, Leino R. *Declaring and Checking Non-null Types in an Object-Oriented Language.* OOPSLA'2003, 302–312.

[9] Cousot P, Cousot R. *Static determination of dynamic properties of programs.* In 2nd Int. Symp. on Programming, Dunod, Paris, 1976.

[10] Gupta R. *A fresh look at optimizing array bound checking.* In PLDI 1990, pages 272–282, June 1990.

[11] Chin WN, Goh EK. *A reexamination of "Optimization of array subscripts range checks".* TOPLAS 1995, Vol. 17, No 2, 217–227.

[12] Flanagan C, Qadeer S. *Predicate abstraction for software verification.* POPL 2002, pages 191–202, Portland, January 2002.

[13] Lahiri SK, Bryant RE, Cook B. *A symbolic approach to predicate abstraction.* CAV 2003, pages 141–153, LNCS 2725.

[14] Blanchet B, Cousot P, Feret J, Mauborgne L, Miné A, Monniaux D, Rival X. *A static analyser for large safety-critical software.* PLDI 2003.

[15] Lahiri SK, Bryant RE. *Indexed predicate discovery for unbounded system verification.* CAV 2004, pages 135–147, LNCS 3114, 2004.

[16] Bradley AR, Manna Z, Sipma HB. *What's decidable about arrays?* VMCAI 06, pages 427–442. LNCS 3855, Springer Verlag, 2006.

[17] Beyer D, Henzinger TA, Majumdar R, Rybalchenko A. *Path invariants.* PLDI 2007, pages 300–309, San Diego (CA), June 2007.

[18] Jhala R, McMillan KL. *Array abstraction from proofs.* CAV 2007, pages 193–206, LNCS 4590, Springer Verlag, 2007.

[19] Iosif R. Habermehl P, Vojnar T. *What else is decidable about arrays?* In R. Amadio, editor, FOSSACS 2008. LNCS, Springer Verlag, 2008.

[20] Halbwachs N, Péron M. *Discovering Properties about Arrays in Simple Programs.* PLDI 2008, pages 339–352, Tucson (AZ), June 2008.

[21] Ellis MA, Stroustrup B. *The Annotated C++ Reference Manual.* "Addison-Wesley, Reading, Massachusetts", 1990

[22] Driesen K, Hölzle U. *The Direct Cost of Virtual Function Calls in C++.* OOPSLA'96, ACM Press, 1996, 306–323.

[23] Sonntag B, Colnet D. *Lisaac: the power of simplicity at work for operating system.* TOOLS Pacific'2002, Sydney, Australia", 45–52.

[24] Zendra O, Colnet D. *Adding external iterators to an existing Eiffel class library.* TOOLS Pacific'99, Melbourne, Australia, pages 188–199.

[25] Jones R, Lins R. *Garbage Collection.* "Wiley", 1996, ISBN 0-471-94148-4w

[26] Colnet D, Coucaud P, Zendra O. *Compiler Support to Customize the Mark and Sweep Algorithm.* ISMM'98, 154–165.