Université de Montréal

Étude de cas sur l'ajout de vecteurs d'enregistrements typés dans Gambit Scheme

par
Benjamin Cérat

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en informatique

Août, 2014

© Benjamin Cérat, 2014.

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé:

Étude de cas sur l'ajout de vecteurs d'enregistrements typés dans Gambit Scheme

présenté par:

Benjamin Cérat

Mémoire accepté le: . . . .

RÉSUMÉ

Dans le but d'optimiser la représentation en mémoire des enregistrements Scheme dans le compilateur Gambit, nous avons introduit dans celui-ci un système d'annotations de type et des vecteurs contenant une représentation abrégée des enregistrements. Ces derniers omettent la référence vers le descripteur de type et l'entête habituellement présents sur chaque enregistrement et utilisent plutôt un arbre de typage couvrant toute la mémoire pour retrouver le vecteur contenant une référence.

L'implémentation de ces nouvelles fonctionnalités se fait par le biais de changements au *runtime* de Gambit. Nous introduisons de nouvelles primitives au langage et modifions l'architecture existante pour gérer correctement les nouveaux types de données. On doit modifier le *garbage collector* pour prendre en compte des enregistrements contenants des valeurs hétérogènes à alignements irréguliers, et l'existence de références contenues dans d'autres objets. La gestion de l'arbre de typage doit aussi être faite automatiquement.

Nous conduisons ensuite une série de tests de performance visant à déterminer si des gains sont possibles avec ces nouvelles primitives. On constate une amélioration majeure de performance au niveau de l'allocation et du comportement du *gc* pour les enregistrements typés de grande taille et des vecteurs d'enregistrements typés ou non. De légers surcoûts sont toutefois encourus lors des accès aux champs et, dans le cas des vecteurs d'enregistrements, au descripteur de type.

Mots clés: Compilation, langage dynamique, gestion de mémoire, arbre de typage, optimisation.

ABSTRACT

In order to optimize the in memory representation of Scheme records in the Gambit compiler, we introduce a type annotation system on record fields. We also introduce flat vector of records containing an abbreviated representation of those records. These vectors omit the header and reference to the type descriptor on contained records and use a type tree spanning the whole memory to recover the type as needed from an internal pointer.

The implementation of the new functionnalities is done through changes in the Gambit *runtime*. We add new primitives to the language and modify the existing architecture to correctly handle the new data types in a way transparent that is transparent to the user. To do so, we modify the garbage collector to account to account for the existance of internal references and of heterogenous records whose fields may not be alligned to a word and need not be boxed. We also have to automatically and systematically update hte type tree to reflect the live vectors.

To asses our implementation's performance, we run a serie of benchmarks. We measure significant gains on allocation time and space with both typed records and contained records. We also measure a minor overhead in access costs on typed fields and major loss on accesses to the type descriptor of contained records.

Keywords: Compilation, dynamic programming language, memory management, type trees, optimization.

TABLE DES MATIÈRES

RÉSUMÉ . . . . **iii**

ABSTRACT . . . . **iv**

TABLE DES MATIÈRES . . . . **v**

LISTE DES TABLEAUX . . . . **viii**

LISTE DES FIGURES . . . . **ix**

LISTE DES SIGLES . . . . **xiii**

CHAPITRE 1 : INTRODUCTION **1**

1.1 Scheme 1

1.2 Gambit 2

1.2.1 Taggage 4

1.3 Représentations des enregistrements 6

CHAPITRE 2 : SYSTÈME DE BASE **9**

2.1 Allocation 9

2.1.1 La mémoire 9

2.1.2 Movable 10

2.1.3 Still 10

2.1.4 Perm 11

2.2 *Garbage collector* 11

2.2.1 Stop-and-copy 12

2.2.2 Mark-and-sweep 12

2.2.3 Compteur de références 13

2.2.4 Racines 13

2.2.5 Garbage_collect 13

2.2.6 Mark_continuation 15
2.2.7 Mark_rc 15
2.2.8 Mark_array 15
2.2.9 Scan 17
2.3 Vecteurs 17
2.3.1 Allocation 17
2.3.2 Référence 19
2.3.3 Mutation 20
2.3.4 Fonctions utilitaires 21
2.4 Types 22
2.4.1 Définition 22
2.4.2 Héritage 24
2.4.3 Descripteur de type 25
2.4.4 Define-type-expand 26
2.4.5 Test de typage 30
2.4.6 Construction 31
2.4.7 Références et mutations 32

CHAPITRE 3 : ENREGISTREMENTS CONTENANT DES CHAMPS TYPÉS STATIQUEMENT 34
3.1 Survol 34
3.2 Modifications à *define-type* 35
3.3 Primitive d'allocation 43
3.4 Primitives d'accès et de mutations 43
3.5 Optimisation *inline* 46
3.6 Modifications au *garbage collector* 50

CHAPITRE 4 : VECTEURS D'ENREGISTREMENTS 55
4.1 Survol 55
4.2 Arbre de conteneurs 56
4.2.1 Paramètres 57

4.2.2 Initialisation 62
4.2.3 Ajout 62
4.2.4 Retrait 66
4.3 Enregistrements contenus 69
4.4 Allocations 72
4.5 Références et mutations 74
4.6 Modifications au *garbage collector* 80

CHAPITRE 5 : ÉVALUATION 83
5.1 Méthodologie 83
5.2 Déclarations 83
5.3 Enregistrements typés 86
5.3.1 Programmes de tests 86
5.3.2 Résultats 88
5.4 Vecteurs d'enregistrements 93
5.4.1 Programmes de tests 93
5.4.2 Résultats 97
5.5 Combinaisons 99
5.5.1 Programmes de tests 99
5.5.2 Résultats 100
5.6 Travaux connexes 101
5.6.1 Chez Scheme 102
5.6.2 Allocation par régions 102
5.6.3 BIBOP 103
5.6.4 Table de pagination multiniveau 104
5.6.5 GC de Boehm 104

CHAPITRE 6 : CONCLUSION 105
6.1 Enregistrements typés 105
6.2 Vecteurs d'enregistrements 106
6.3 Résultats 107

LISTE DES TABLEAUX

5.I Résultats avec des enregistrements typés 88
5.II Résultats avec des enregistrements typés 92
5.III Résultats avec des vecteurs d'enregistrements 98
5.IV Résultats avec des vecteurs d'enregistrements typés 100

LISTE DES FIGURES

1.1 Structure de Gambit 3
1.2 Représentation en bits des objets 4
1.3 Représentation en bits des entêtes 5
1.4 Exemple d'utilisation de *define-type* 6
1.5 Représentation des enregistrements 6
1.6 Vecteur d'enregistrements 7
1.7 Enregistrements avec des champs de 32 bits 7

2.1 Pseudocode de *garbage_collect()* 14
2.2 Pseudocode de *mark_array* 16
2.3 Code C de *##vector* 18
2.4 Macros formant un appel à *##vector* 19
2.5 Spécification de l'extension *inline* de *##vector-ref* 20
2.6 Code C généré pour *##vector-ref* 20
2.7 Code généré pour *##vector-set !* 21
2.8 Code généré pour *##vector ?* 21
2.9 Code généré pour *##vector-length* 22
2.10 Exemple de définition d'un type *point* 22
2.11 Expansion d'un type *point* 23
2.12 Exemple de définition d'un *point* comme type parent 24
2.13 Représentation d'un point 2d 25
2.14 *##type-type* 25
2.15 Définition de *define-type* 26
2.16 Paramètres de l'expansion de *define-type* 27
2.17 Fonction générant les champs 28
2.18 *generate-initializations* 29
2.19 Expansion du constructeur avec une valeur initiale 29
2.20 Construction de la fonction d'accès 30

2.21 Construction de la macro d'accès 30
2.22 Expansion de tests de type 31
2.23 Expansion du constructeur 31
2.24 Primitive *##structure* 32
2.25 Primitive *##structure-ref* 33
2.26 Extension *inline* de *##unchecked-structure-ref* 33

3.1 Enregistrements avec des champs de tailles hétérogènes 35
3.2 Enregistrements multiniveaux avec *padding* 35
3.3 Exemple de définition d'un point typé 36
3.4 Primitive *##sizeof-typed-field* 36
3.5 Calcul des paramètres requis pour les champs typés 38
3.6 Extraction des paramètres supplémentaires requis pour les champs typés 39
3.7 Définition du descripteur de type statique 40
3.8 Expansion des mutateurs et des accesseurs dans *##define-type-expand* 40
3.9 Construction des paramètres des champs 41
3.10 Construction des appels aux accesseurs 42
3.11 Construction d'un point 1d 8 bits 43
3.12 Construction de l'appel au constructeur *make-<type>* 44
3.13 Primitive *##typed-structure* 45
3.14 Primitive *##unchecked-int8-structure-ref* et *set!* 46
3.15 Code généré pour les accesseurs et mutateurs 47
3.16 *Unboxing* des entiers signés de 32 bits et non signés de 64 bits . . 48
3.17 Spécialisation des primitives d'accès 48
3.18 Primitives d'accès et de mutations 49
3.19 Définition de l'expansion *inline* de *##typed-structure* 50
3.20 Code générant *##typed-structure inline* 51
3.21 Macros servant à générer l'expansion de *##typed-structure* 52
3.22 *##type-type* étendu 53

3.23 Branche pour les enregistrements dans *scan* 53
3.24 Fonction *scan_struct* 54

4.1 Vecteur d'enregistrement 55
4.2 Arbre de conteneurs 57
4.3 Paramètres de l'arbre de conteneurs 58
4.4 Variables globales dans mem.c 58
4.5 Constantes liées à l'arbre de conteneurs 59
4.6 Macros liées à l'arbre de conteneurs 60
4.7 Macros spécifiques en 32 bits 60
4.8 Macros spécifiques en 64 bits 61
4.9 Macros parcourant l'arbre de conteneurs 61
4.10 Fonction *init_type_tree* 62
4.11 Fonction *add_container* 63
4.12 Fonction *add_type_block* 65
4.13 Fonction *free_container* 67
4.14 Fonction *check_branching* 68
4.15 Fonction *free_type_block* 69
4.16 Primitive *##structure-type* 70
4.17 Code généré pour *##contained ?* 71
4.18 Primitive *##structure-direct-instance-of ?* 71
4.19 Code généré pour *##structure-type* et *##structure-direct-instance-of ?* 71
4.20 Primitive *##structure-vector ?* 72
4.21 Code généré pour *##structure-vector ?* 72
4.22 Exemple d'expansion de *make-point-vector* 73
4.23 Construction de *make-<type>-vector* 74
4.24 Primitive *##structure-vector* 75
4.25 Fonction *alloc_container* 76
4.26 Construction des définitions pour *<type>-vector-ref* 77
4.27 Code généré pour *##structure-vector-ref* 77

4.28 Primitive *##structure-vector-set !* . . . 78
4.29 Génération de l'expansion *inline* de *##structure-vector-set !* . . . 79
4.30 Mutateur d'un élément *float32* dans un *##structure-vector-set !* . . 80
4.31 Branche des vecteurs d'enregistrements dans *scan* . . . 81
4.32 Fonction *scan_container* . . . 82

5.1 Déclarations au début de chaque test. . . . 84
5.2 Code de *##structure-typed* . . . 85
5.3 Coeur de la boucle de *##structure-typed* . . . 85
5.4 Code de *##structure* . . . 87
5.5 Code de *##structure-typed* . . . 87
5.6 Code de *distance-sort int32* . . . 89
5.7 *VECTOREF* vs *STRUCTREF* . . . 91
5.8 *VECTORSET* vs *STRUCTSET* . . . 91
5.9 *S8VECTORREF* vs *INT8STRUCTREF* . . . 92
5.10 *S32Unbox* . . . 92
5.11 *##structure* avec un vecteur . . . 94
5.12 *##structure* avec un vecteur d'enregistrements . . . 95
5.13 *##structure-type* avec vecteur . . . 95
5.14 *##structure-type* contenu . . . 96
5.15 *CONTAINERREF* . . . 97
5.16 Conversion vers les programmes typés . . . 100
5.17 Utilisation de *define-ftype* . . . 103

LISTE DES SIGLES

FFI	Foreign funtion interface
gc	Garbage collector
gsc	Gambit Scheme compiler
gsi	Gambit Scheme interpreter
Lisp	List processing

CHAPITRE 1

INTRODUCTION

Il n'est pas possible, en général, de déterminer au moment de la compilation d'un programme la quantité de mémoire qu'il consommera puisqu'elle dépend des entrées sur lequel le programme sera exécuté. Une solution très simple consiste à allouer une quantité fixe assez grande pour couvrir les cas souhaités et lancer une erreur s'il en manque (cette approche utilise évidemment plus de mémoire que nécessaire dans les cas usuels). Une autre approche (celle de C entre autres) demande la mémoire au système d'exploitation dynamiquement au besoin lors de l'exécution. La majorité des langages de programmation moderne automatisent l'allocation et la récupération de la mémoire lors de l'exécution des programmes par le biais de routines exposées dans l'environnement d'exécution standard du langage. Dans le cas de langages typés dynamiquement, il est assez simple d'avoir une récupération précise de la mémoire inutilisée, car l'information de typage est disponible au moment de l'exéc ution ce qui permet de distinguer les données numériques des références en mémoires sans ambiguïtés. Toutefois, la nécessité d'encoder cette information consomme de la mémoire.

Nous traiterons ici d'une approche permettant une gestion mémoire efficace pour les enregistrements de Scheme dans le système Gambit. Cette approche implique l'ajout de nouvelles primitives au dialecte et donc des modifications au *runtime* de Gambit.

1.1 Scheme

Scheme [**?**] est un langage de programmation multiparadigme typé dynamiquement, à portée lexicale et à liaison tardive, dérivé de Lisp. Traditionnellement, l'implémentation peut se faire avec un compilateur statique ou un interprète.

Le langage supporte des concepts avancés comme des continuations et des fermetures de première classe, la gestion de mémoire automatique, les macros et l'évaluation dynamique de code. Les entiers ont une précision infinie. L'optimisation des appels

terminaux est requise par la spécification du langage et permet l'utilisation de la récursion pour faire des itérations ce qui permet l'utilisation efficace du style de programmation fonctionnelle. Plusieurs structures de données sont supportées, notamment les vecteurs, les listes chaînées, des tables de hachage et des enregistrements analogues aux *struct* C, mais supportant une forme d'héritage simple.

Les implémentations de Scheme comportent typiquement une gestion de mémoire automatique qui utilise l'encodage des données pour déterminer précisément la vivacité des objets en mémoire (par opposition, par exemple, à l'algorithme de Boehm [**?** , p. 166]). L'information de typage étant nécessaire à l'exécution et à la gestion mémoire, elle doit être conservée à même les valeurs. Souvent, cela se fait en utilisant des bits de poids faible laissés vacants par l'alignement des blocs de mémoire alloués (ceux-ci étant typiquement alignés sur des multiples de 4 ou de 8) et dans un mot d'entête précédant la donnée en mémoire. La récupération se fait typiquement en parcourant chaque valeur accessible depuis un ensemble de racines (la pile d'exécution, les registres et les variables globales en général) et en suivant récursivement les références ainsi obtenues (distinguables par leur *tag*).

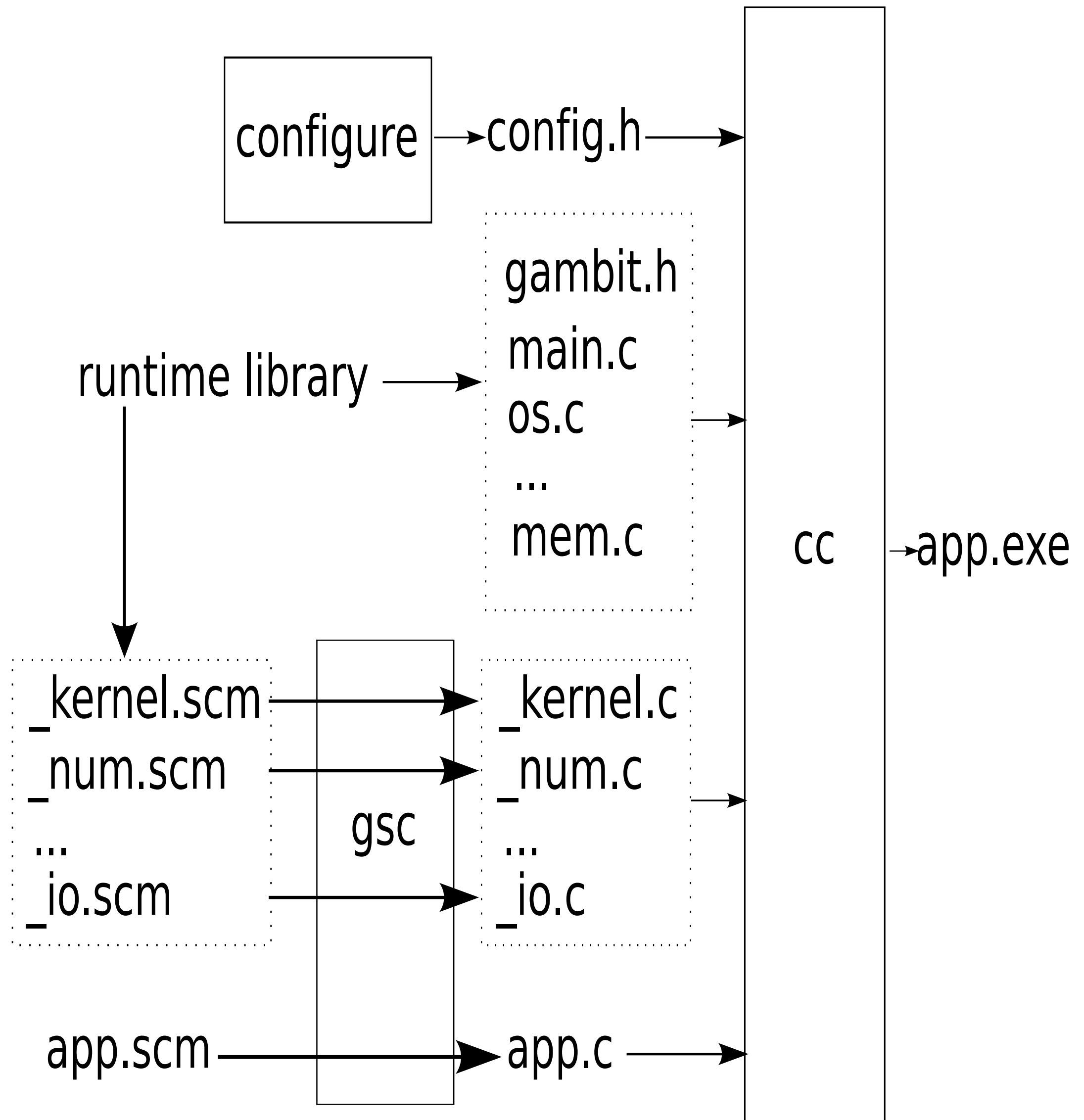
1.2 Gambit

Le compilateur Gambit[**?**] (Gambit Scheme Compiler ou gsc) est une implémentation de Scheme traduisant vers C. L'utilisation de C comme intermédiaire assure la compatibilité interplateforme. Le compilateur offre la possibilité de générer statiquement des exécutables ou un fichier C (figure 1.1). Un interprète (gsi) est aussi disponible, qui excelle au débogage de programme au prix d'une certaine lenteur.

Gambit présente, en plus des primitives spécifiées dans le standard, un ensemble d'extensions incluant des vecteurs de données numériques homogènes de taille fixe.

La partie du *runtime* écrite en C gère les primitives de bas niveau de la machine virtuelle de Gambit. Ces fonctions permettent une implémentation efficace dans un langage de bas niveau et sont appelées par les programmes Scheme au moyen d'une interface (FFI) permettant d'exécuter du code C. Les fonctions primitives sont dénotées

Figure 1.1 – Structure de Gambit



par le préfixe ## et utilise des primitives comme *c-lambda* ou *c-code* qui permettent de citer directement du code C pour appeler les routines du *runtime*. C'est principalement cette partie qui sera étendue pour implémenter notre extension.

Ce runtime inclut un fichier d'entête (*gambit.h*) contenant des macros et des définitions pour tous les paramètres dépendant de la machine d'exécution. Entre autres, on y retrouve les calculs d'accès aux structures de données, les *tags* et sous-types et les déclarations de toutes les fonctions C exposées par le *runtime*. Plusieurs macros C servent à dénoter les différents types de fonctions dans le *runtime*. Par exemple, le préfixe *___HIDDEN* devant une fonction dénote que celle-ci ne sera pas exportée en dehors du fichier où elle se trouve. *___EXP_FUNC* quant à elle dénote que la fonction sera visible au code Scheme.

Le fichier *mem.c* contient toutes les primitives de gestion de mémoire, incluant le *garbage collector*.

Figure 1.2 – Représentation en bits des objets

1.2.1 Taggage

Comme l'allocation est toujours faite de façon à ce que le pointeur retourné soit aligné sur un multiple de 4 ou 8, les deux derniers bits valent toujours 0 (figure 1.2). Ils sont donc utilisés pour placer des informations de typage minimales sur chaque pointeur afin de pouvoir garantir la précision du *garbage collector*.

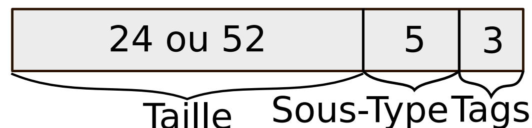
- fixnum (0) : De petits entiers d'au plus 30 bits.
- special (2) : Des valeurs constantes prédéfinies et des caractères
- subtyped (1) : Des pointeurs vers des objets alloués en mémoires (autres que les paires).
- pair (3) : Des pointeurs vers des paires allouées en mémoires.

Pour des raisons de performance arithmétique, les valeurs entières de petite taille sont étiquetées à 0, ce qui permet de faire les opérations directement sans avoir d'étapes supplémentaires. Logiquement, le second type de données immédiates a lui aussi une étiquette avec les bits de poids faibles égaux à 0, ce qui permet de simplifier les tests servant à déterminer si la valeur se trouve en mémoire. Les deux valeurs restantes servent à représenter des pointeurs. Les paires étant un type très commun, elles sont séparées afin de simplifier les tests de typages.

Un certain nombre d'informations sont nécessaires au *garbage collector* afin de pouvoir parcourir les données. Celles-ci sont placées dans un mot immédiatement avant la donnée vers laquelle le pointeur réfère ce qui permet de s'assurer de pouvoir le retrouver que ce soit en parcourant linéairement une zone de mémoire ou en suivant un pointeur.

Parmi les informations requises, on a le nombre de mots dans l'objet, son sous-type et le genre d'allocation. L'entête est donc subdivisé en trois champs : 3 bits pour le genre, 5 bits pour le sous-type et le reste du mot pour la taille (24 bits en 32 bits et 56 bits en 64 bits). La figure 1.3 montre la distribution de ces champs dans un mot.

Figure 1.3 – Représentation en bits des entêtes



Le genre d'allocation peut-être :

- MOVABLE0 (0) : Les objets déplaçables de génération 0.
- FORW (3 et 7) : Les objets ayant été déplacés. Accompagne un pointeur vers leur nouvel emplacement.
- STILL (5) : Les objets qui ne sont pas déplacés lors d'une collection.
- PERM (6) : Les objets qui sont permanents et ne sont jamais collectés.

Les genres 1, 2 et 4 sont présentement inutilisés, mais sont réservés dans l'optique de potentiellement rendre le *garbage collector* générationnel. Le genre *FORW* correspond à

deux valeurs, car on réutilise l'entête pour le pointeur vers la nouvelle adresse et celui-ci utilise le 3e bit qui peut donc être 0 ou 1.

1.3 Représentations des enregistrements

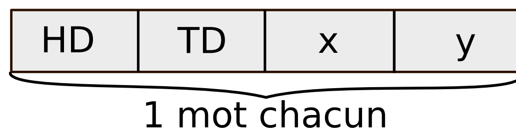
Il est possible de définir des types à l'aide de la macro *define-type* (figure 1.4).

Figure 1.4 – Exemple d'utilisation de *define-type*

```
(define-type point x y)
```

Ceux-ci sont utilisés pour exprimer des données contenant une série de champs nommés ayant une représentation fixe (figure 1.5). Les utilisations de ces objets typés vérifient que la méthode accède bien à un objet du bon type au moment de l'exécution, ce qui nécessite de stocker une référence vers le descripteur de type dans l'objet. Ce champ supplémentaire s'ajoute à l'entête usuel trouvé sur tous les objets Scheme. Cela engendre une utilisation de mémoire supplémentaire substantielle lorsque comparé aux structures C. En effet, pour un enregistrement de N champs, on occupera $2+N$ mots machines et pour M enregistrements, $M(2+N)$ mots.

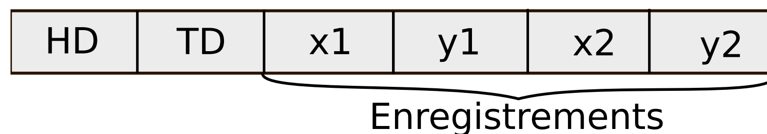
Figure 1.5 – Représentation des enregistrements



Pour réduire l'espace nécessaire, il est envisageable d'allouer en bloc les objets partageant le même type et de réutiliser une référence commune au descripteur de type (figure 1.6). Cela permettrait du même coup d'avoir seulement un entête pour le bloc. Celui-ci n'occuperait donc que $2+MN$ mots pour stocker M instances d'enregistrements de N champs. Il devient toutefois obligatoire d'introduire une mécanique pour s'assurer de ne jamais traiter les enregistrements contenus dans ces blocs comme des objets individuels puisque leur entête et leur descripteur de type ne se trouvent pas à l'endroit

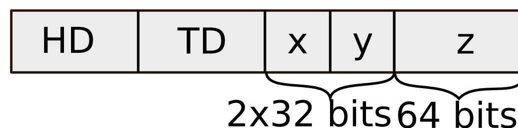
usuel et qu'on veut que l'utilisation de ces enregistrements soit complètement transparente. Il devient aussi nécessaire d'avoir un moyen de retrouver le descripteur de type à un décalage arbitraire. Pour ce faire, on utilise un marqueur distinct sur les références des enregistrements contenus et on introduit un arbre de typage couvrant toute la mémoire et indexé par l'adresse de la référence.

Figure 1.6 – Vecteur d'enregistrements



De plus, chacun des champs est alloué de façon à pouvoir contenir n'importe quel objet Scheme alors qu'on utilise en général une définition de type pour des objets ayant un contenu précis qui a fréquemment une nature numérique. Dans l'optique de compacter la représentation, il pourrait être souhaitable de pouvoir donner un type essentiellement statique aux champs au moment de la définition du type (figure 1.7).

Figure 1.7 – Enregistrements avec des champs de 32 bits



Ces annotations de types permettraient d'avoir des types numériques à précision fixe autre que ceux correspondant à la taille d'un mot moins les bits de marquage. Pour des données numériques de petite taille, un gain substantiel peut être accompli en remplaçant par exemple chaque objet Scheme de 64 bits par 8 entiers de 8 bits chacun.

La diminution de la taille des données combinée à l'allocation en masse des enregistrements devrait réduire considérablement la pression sur le *garbage collector* et permettre plus de flexibilité au programmeur souhaitant éviter son déclenchement durant des parties sensibles du code. Il n'est toutefois pas garanti que les gains réalisés vont compenser les coûts de performance liés aux méthodes d'accès, de mutations et de tests de typages plus complexes ainsi que les coûts liés à la gestion de l'arbre de typage.

Dans le second chapitre, nous introduisons l'implémentation initiale des enregistrements modifiés dans le compilateur Gambit Scheme. Dans le troisième, nous traiterons des changements requis à l'implémentation des champs typés dans les enregistrements. Dans le quatrième, nous traiterons de l'ajout de vecteurs d'enregistrements au compilateur et dans le dernier chapitre nous ferons une évaluation de performance des changements apportés.

CHAPITRE 2

SYSTÈME DE BASE

Ce chapitre introduit la structure et le fonctionnement du compilateur Gambit avant les modifications faites lors de l'implémentation des vecteurs de structures et des structures typées afin de mettre en contexte les changements qui seront décrits dans les chapitres suivants.

2.1 Allocation

L'allocation des objets dans Gambit se fait selon l'une de trois approches : *movable*, *still* et *permanent*.

Les objets alloués comme *movable* sont, comme le nom l'indique, déplaçables lors des opérations de récupération de la mémoire. Ils sont placés dans une sous-section du *tas* ayant une taille variable. Chaque objet alloué est constitué d'un ensemble contigu de mots machines (de 32 ou 64 bits) précédé d'un mot d'entête notant l'information de base nécessaire au *runtime*. Chacun de ces objets est référé par un ou plusieurs pointeurs portant une étiquette permettant de différencier les objets alloués dans le tas des valeurs numériques ou des valeurs spéciales comme *faux* ou *'()* (la liste vide).

2.1.1 La mémoire

La mémoire est assignée au programme par le *runtime*. La taille allouée fluctue en fonction de l'utilisation de mémoire du programme et sa taille initiale peut être fixée par un argument donné à l'exécutable.

Un ensemble de variables dans le *runtime* subdivise l'espace total en plusieurs catégories, soit les *movable0*, les *still* et les *perm*. Une liste des sections utilisables pour l'allocation de *movable* est conservée et subdivisée en sections correspondant au tas et à la pile. Les *stills* sont conservés dans une liste chaînée qui permet de les parcourir facilement et sont alloués individuellement. Les *perms* sont alloués sur le tas C et ne sont

pas pris en compte par le reste du programme.

2.1.2 Movable

La part de la mémoire assignée aux *movables* est subdivisée en plusieurs sous-parties appelées des *msections*. Celles-ci sont constituées de sections de mémoire liées les unes aux autres dans une liste doublement chaînée ordonnée par adresses. Ces *msections* sont implémentées sous forme d'une *struct* C.

Elles sont chacune séparées en deux parties de taille égales et l'allocation n'est faite que dans l'une des moitiés afin de garantir que suffisamment d'espace pour une collection soit disponible lorsque le *garbage collector* est appelé. L'information sur la partie à utiliser est stockée dans une variable globale du *runtime* : *tospace_at_top*.

D'autres variables désignent les zones servant de tas et de pile ainsi que les pointeurs d'allocation courants. Le tas sert généralement à contenir tous les objets sauf les blocs d'activation des continuations nouvellement allouées qui se trouvent sur la pile.

L'allocation sur le tas se fait de l'adresse la plus petite vers la plus grande et l'inverse sur la pile. Il est donc possible d'avoir les deux structures dans la même *msection* si l'espace assigné est trop restreint pour les séparer. En général, l'allocation d'un *movable* se fait par un simple déplacement du pointeur approprié (conservé dans le *processor state* de la machine virtuelle). Un test est fait au début de chaque bloc de base pour s'assurer qu'un espace suffisant soit disponible et une collection est lancée si ce n'est pas le cas.

2.1.3 Still

Les objets de type *still* sont alloués individuellement dans des blocs de mémoires avec un appel à *malloc*. Une série de champs sont ajoutés avant l'entête : un lien vers l'objet *still* suivant, un champ de compteur de références qui dénombre le nombre de références à l'objet, la taille de l'objet et un champ servant à chaîner les *stills* ayant été marqué durant une collection.

L'allocation de ce type d'objets se fait par le truchement de la fonction *alloc_scmobj*. Celle-ci vérifie que l'espace nécessaire est disponible et le crée si ce n'est pas le cas,

alloue le bloc de mémoire et initialise les champs internes.

2.1.4 Perm

Les objets de type *perm* sont alloués sur le tas C et conservés dans des variables globales. Ils ne peuvent contenir que des pointeurs vers d'autres structures permanentes et ne sont donc jamais parcourus par le *garbage collector*.

2.2 *Garbage collector*

La gestion de la mémoire dans Gambit est entièrement faite de façon automatique par un algorithme de collection. Des vérifications sont insérées au début des blocs de base et dans les fonctions d'allocations d'objets non déplaçables pour s'assurer de déclencher la fonction de récupération de mémoire lorsque c'est nécessaire.

L'espace potentiellement libéré se trouve dans les sections *still* et *movable*. Un algorithme de type *stop-and-copy* (**?** , p. 43]) est utilisé pour les données qui peuvent être déplacées, car l'inclusion de la pile dans cette section et le style de programmation fonctionnel prévalant font en sorte d'alloué beaucoup d'objets ayant une courte durée de vie ce qui rend un algorithme ayant un coût proportionnel à la taille des données vivantes très attrayant, de même que la compaction naturelle du tas.

Évidemment, les objets non déplaçables ne peuvent être gérés de cette façon. Un algorithme de *mark-and-sweep* (**?** , p. 31]) hybridé avec un compteur de références (**?** , p. 57]) est donc employé dans ce cas. Celui-ci parcourt les objets en suivant les liens et ajoute tous les objets *stills* dans une liste d'objets marqués. Ensuite, il libère tous ceux qui n'ont ni marquage ni références depuis du code C. L'étape de comptage de références est nécessaire, car les fonctions d'allocations écrites en C ne conservent pas de liens Scheme valides, mais peuvent déclencher le *garbage collector*.

Comme les phases de marquage et de copie sont intrinsèquement reliées, la traversée de toutes les régions doit être faite à chaque collection.

2.2.1 Stop-and-copy

Cette partie de la collection est faite en utilisant une version modifiée de l'algorithme de Cheney. L'espace collecté est divisé en une série de *fromspace* et de *tospace*. Un ensemble de racines est parcouru et les objets dans le *fromspace* qui sont rencontrés sont copiés séquentiellement dans leur *tospace* correspondant. La version initiale dans le *fromspace* est réétiquetée comme étant un *FORW* et une référence vers la nouvelle adresse est mise dans l'entête. Si on rencontre un objet déjà copié, on met à jour la valeur de la référence.

Après avoir visité les racines, on parcourt les *tospaces* de façon linéaire en mettant à jour chacune des références rencontrées et en copiant les nouveaux objets à la fin du *tospace*. Une fois à la fin, les objets vivants ont tous été recopiés et le *fromspace* devient le prochain *tospace* en modifiant la variable *tospace_at_top*.

Une certaine attention doit être prêtée au traitement des références faibles qui sont utilisées dans certaines structures de données, tels les testaments et les tables de hachage. Les testaments sont des fonctions qui sont exécutées à un moment indéterminé après la récupération de l'objet auquel il est attaché par le biais d'une référence faible.

En effet, elles ne doivent en général pas être suivies, mais les testaments doivent être conservés jusqu'à leurs exécutions lorsqu'un objet est récupéré. Pour ce faire, on parcourt les données une fois sans considérer les références faibles et en notant les testaments rencontrés, puis on recommence en traversant les références faibles pour déterminer les objets morts dont les testaments sont vivants et donc à exécuter.

2.2.2 Mark-and-sweep

Le *mark-and-sweep* se fait de façon interreliée avec le *stop-and-copy*. Le parcours des racines crée une liste d'objets *still* à traiter. Ceux-ci sont ensuite visités un par un. Lorsqu'une référence à un objet *still* est rencontrée, on l'ajoute aux objets à traiter et on le chaîne aux autres objets qui ont été marqués.

Une fois la collection terminée, on parcourt l'ensemble des objets still et on récupère ceux qui ne sont ni marqués ni référencés par un objet permanent.

2.2.3 Compteur de références

Le compteur de références n'est quant à lui utilisé que durant l'allocation d'un objet afin de s'assurer que la référence vers celui-ci a eu le temps d'être placée dans une variable Scheme avant une collection. Tous les objets doivent être relâchés avec la fonction *release_scmobj* lorsque le code C qui les alloue se termine.

2.2.4 Racines

Le *garbage collector* utilise un ensemble de racine comme point de départ pour déterminer quels objets sont vivants. Cet ensemble comprend notamment l'ensemble des registres et des variables globales, ainsi que les continuations sur la pile d'exécution et les objets *still* ayant un compteur de références non nul.

2.2.5 Garbage_collect

La fonction *garbage_collect* (figure 2.1) est celle appelée lorsqu'une collection est nécessaire. Elle contient donc l'ensemble de la logique de collection. Par le biais de la fonction *mark_array*, elle visite les registres et les variables globales puis utilise *mark_continuation* et *mark_rc* pour visiter les blocs d'activations des continuations de la pile et les objets *still* ayant un compte de références d'au moins 1. Ceci donne l'ensemble des racines qui sont considérées vivantes par défaut.

Ensuite, on parcourt les objets récupérables sans suivre les références faibles, on gère les testaments et on recommence en les considérant comme des références fortes.

On utilise ensuite la fonction *process_gc_hash_tables* pour parcourir les tables de hachage et les marquer comme potentiellement salie. Ce marquage va provoquer un *rehashing* paresseux au moment de l'utilisation de la table ce qui permet d'éviter de recalculer inutilement les positions relatives des clefs.

Une fois l'ensemble des objets vivants parcourus, on élimine tous les *stills* qui n'ont pas été rejoints avec la fonction *free_unmarked_still_object* et on calcule la taille de la mémoire nécessaire au programme. Avant de compacter ou agrandir le tas, on bouge la pile si sa *msection* n'est plus requise. Si le changement du tas n'est pas possible, on lance

Figure 2.1 – Pseudocode de *garbage_collect()*

```
garbage_collect(processor_state, words_needed) :
    /* Assuming processor_state as
        an argument to every functions */

    alloc_stack_ptr = processor_state->fp
    alloc_heap_ptr = processor_state->hp

    flip_tospace_and_fromspace()

    reached_gc_hashtables = 0
    init_still_obj_to_scan()

    scan_registers()
    scan_global_var()
    mark_continuation()
    mark_rc()

    traverse_weak_ref = 0

again:
    scan_still_obj_to_scan()
    scan_movable_obj_to_scan()

    process_wills()
    if traverse_weak_ref == 0 :
        traverse_weak_ref = 1
        goto again

    process_gc_hashtable()
    free_unmarked_still_obj()
    size = new_heap_size()
    move_stack()
    resize_heap(size)

    prepare_processor_state()
    raise_gc_interrupt()
    return overflow
```

une exception. Finalement, on calcule certaines propriétés sur la collection et on met à jour l'état du processeur dans une structure de *pstate*, on lève l'interruption du *gc* et on retourne l'état de l'*overflow*.

2.2.6 Mark_continuation

La fonction *mark_continuation* traverse l'ensemble des continuations sur la pile et leur chaînage lorsqu'elles sont transférées sur le tas pour marquer toutes les références se trouvant dans leurs blocs d'activations. La fonction *mark_array* est encore une fois utilisée sur les champs qui doivent être marqués.

2.2.7 Mark_rc

La procédure *mark_rc* parcourt simplement la liste des *stills* ayant un compteur de références d'au moins 1 en appelant *mark_array* dessus.

2.2.8 Mark_array

La procédure *mark_array* (figure 2.2) contient la majeure partie de la logique de collection. Elle s'occupe de déterminer si la valeur parcourue doit être marquée ou recopiée en fonction de son type et de mettre à jour les références vers des objets déplacés.

Pour ce faire, elle récupère l'entête de l'objet passé en paramètre et en extrait le type et le sous-type à l'aide des macros *HD_TYP* et *HD_SUBTYPE*. Ensuite, si la valeur est déplaçable, il fait la copie dans le *tospace* en respectant les contraintes d'alignement, écrit la nouvelle adresse dans le premier champ de l'objet original et change son étiquette vers un *FORW*.

Dans le cas d'un objet *still*, s'il n'a pas été marqué préalablement, la procédure l'ajoute à la liste *still_objs_to_scan* et change le champ de marquage vers la liste en question.

Si le champ était une sentinelle de redirection, on récupère la nouvelle adresse et on modifie le pointeur vers l'objet qui a été passé en paramètre.

Figure 2.2 – Pseudocode de *mark_array*

```
mark-array(*start, n) :
    get_processor_state()
    *alloc, *limit = alloc_heap_pointer, alloc_heap_limit

    while n > 0 :
        obj = *start

        body = ___UNTAG(obj) + ___BODY_OFS
        head = body[-1]
        subtype = ___HD_SUBTYPE(head)
        head_typ = ___HD_TYP(head)

        if head_typ = ___MOVABLE0 :
            copy(obj, find_next_allocation_spot())
            set_forwarding_pointer(body[-1])

        if head_typ = ___STILL :
            mark(obj)
            chain_to_still_obj_to_scan(body)

        if head_typ = ___FORW :
            *copy_body = ___UNTAG_AS(head, ___FORW) +
                                  ___BODY_OFS
            *start = ___TAG((copy_body -
                          ___BODY_OFS), ___TYP(obj))

        start++
        n-
    update_processor_state()
```

Finalement, le pointeur d'allocation du tas est modifié à une valeur correspondant à celle de la fin d'allocation de copie.

2.2.9 Scan

La fonction scan sert à parcourir les champs des objets en mémoire en évitant de traiter des valeurs quelconques comme des pointeurs. Elle débute en extrayant le sous-type et la taille de l'objet passé en paramètre. Elle fait ensuite un *switch* sur le sous-type et fait un ou des appels à *mark_array* de façon appropriée.

Les types numériques autres que des *fixnums* sont simplement ignorés, de même que les *strings* et les valeurs C. Les références faibles peuvent être des testaments qui sont ignorés si on se trouve dans la première passe de collection. Ils peuvent aussi être un pointeur interne d'une table de hachage.

2.3 Vecteurs

Les vecteurs sont des structures de données Scheme analogues aux tableaux C. Ils sont constitués d'un ensemble de cellules identiques adjacentes en mémoire. Comme Scheme ne permet pas la manipulation directe de pointeurs, un groupe de fonctions primitives est fourni pour couvrir les cas d'utilisations comme l'allocation, l'accès et les mutations aux cellules. Comme tout objet Scheme, les informations de typage et la taille sont stockées dans l'entête. L'étiquette du pointeur est celle d'un objet sous-typé.

Les vecteurs de Gambit sont disponibles pour une variété de types en plus des objets Scheme typiques, soit les entiers signés et non signés entre 8 et 64 bits et les nombres flottants de 32 et 64 bits.

2.3.1 Allocation

Deux fonctions d'allocation font partie des primitives : *##vector* (figure 2.3) et *##make-vector*. La première prend un nombre variable de paramètres et retourne un vecteur contenant exactement ceux-ci comme cellules. L'autre prend le nombre de cellules et optionnellement la valeur d'initialisation et retourne un vecteur de cette taille ayant cette

Figure 2.3 – Code C de *##vector*

```
___SIZE_TS i;
___SIZE_TS n = ___INT(___ARG1);
___SIZE_TS words = n + 1;
___SCMOBJ result;
if (n > ___CAST(___WORD, ___LMASK>>(___LF+___LWS)))
  /* requested object is too big! */
  result = ___FIX(___HEAP_OVERFLOW_ERR);
else if (words > ___MSECTION_BIGGEST)
  {
    ___FRAME_STORE_RA(___R0)
    ___W_ALL
    result = ___EXT(___alloc_scmobj) (___ps, ___sVECTOR, n<<___LWS);
    ___R_ALL
    ___SET_R0(___FRAME_FETCH_RA)
    if (!___FIXNUMP(result))
      ___still_obj_refcount_dec (result);
  }
else
  {
    ___BOOL overflow = 0;
    ___hp += words;
    if (___hp > ___ps->heap_limit)
      {
        ___FRAME_STORE_RA(___R0)
        ___W_ALL
        overflow = ___heap_limit (___PSPNC) &&
                ___garbage_collect (___PSP 0);
        ___R_ALL
        ___SET_R0(___FRAME_FETCH_RA)
      }
    else
      ___hp -= words;
    if (overflow)
      result = ___FIX(___HEAP_OVERFLOW_ERR);
    else
      {
        result = ___TAG(___hp, ___tSUBTYPED);
        ___HEADER(result) = ___MAKE_HD_WORDS(n, ___sVECTOR);
        ___hp += words;
      }
  }
if (!___FIXNUMP(result))
  {
    ___SCMOBJ fill = ___ARG2;
    if (fill == ___ABSENT)
      fill = ___FIX(0);
    for (i=0; i<n; i++)
      ___VECTORSET(result,___FIX(i),fill)
  }
___RESULT = result;
```

valeur dans chaque cellule. Son implémentation est faite par une primitive employant la *FFI*.

Figure 2.4 – Macros formant un appel à *##vector*

```
#define ___BEGIN_ALLOC_VECTOR(n) \
    ___hp[0]=___MAKE_HD_WORDS(n,___sVECTOR);

#define ___ADD_VECTOR_ELEM(i,val)___hp[i+1]=(val);

#define ___END_ALLOC_VECTOR(n)___ALLOC(n+1);

#define ___GET_VECTOR(n)___TAG((___hp-n-1),___tSUBTYPED)
```

La fonction *##vector* a une implémentation optimisée automatiquement par le compilateur vers une séquence d'opérations appropriées (figure 2.4). Pour ce faire, la représentation intermédiaire de l'appel est traduite par *targ-apply-vector* (dans _t-c-2.scm) vers des macros C (définies dans gambit.h.in). Ces opérations variant évidemment en fonction du type de vecteur, la traduction de l'appel est annotée d'un paramètre *kind*. De l'espace est réservé sur le tas avec *targ-heap-reserve-and-check*, *BEGIN_ALLOC_VECTOR* écrit l'entête puis on trouve la position de chacun des arguments sur la pile en faisant le traitement nécessaire s'il s'agit d'un nombre à virgule flottante. Du code écrivant la valeur des variables dans la cellule appropriée est ensuite émis avec *ADD_VECTOR_ELEMENT*. *END_ALLOC_VECTOR* émet ensuite du code décalant le pointeur d'allocation du tas de façon appropriée et *GET_VECTOR* retourne l'adresse du début du vecteur. Des macros analogues sont évidemment disponibles pour les autres types.

2.3.2 Référence

Comme pour *##vector*, les accès aux cellules des vecteurs sont toujours générés de façon *inline* (figure 2.5).

Lorsqu'un appel à *##vector-ref* dans le code compilé est rencontré, le symbole est recherché dans une table de hachage des primitives à *inliner*. La fonction *targ-op* permet

Figure 2.5 – Spécification de l'extension *inline* de *##vector-ref*

```
(targ-op "##vector-ref" (targ-ifjump-apply-u "VECTORREF"))
```

de spécifier que l'expansion correspondant au symbole *##vector-ref* est le résultat de la macro *VECTORREF* (figure 2.6) appliqué sur ses arguments.

Figure 2.6 – Code C généré pour *##vector-ref*

```
#define ___UNTAG_AS(obj,tag)___CAST(___WORD*,(obj)-(tag))

#define ___BODY_AS(obj,tag)(___UNTAG_AS(obj,tag)+ \
        ___BODY_OFS)

#define ___VECTORREF(x,y)*(___WORD*)(((___WORD) \
        ___BODY_AS(x, ___tSUBTYPED))+((y)<<(___LWS-___TB)))
```

Celle-ci prend un pointeur (x) et un décalage (y) et retourne la valeur se trouvant à l'index y du vecteur x. *BODY_AS* prend un pointeur et un sous-type et calcule l'adresse machine auquel le début de l'objet se trouve (en sautant l'entête). Le calcul est fait par une simple soustraction plutôt que par un masque, mais ne peut supporter des opérations sur plusieurs sous-types. L'adresse est une adresse valide C ne comportant plus d'étiquettes, ce qui permet de faire des opérations de pointeurs sur le résultat.

Une fois cette adresse calculée, on *cast* la valeur obtenue en mot machine et on lui ajoute l'index sans les bits d'étiquette. On double le décalage ainsi calculé en 64 bits pour prendre en compte la différence de taille des mots. Après un autre *cast*, vers un pointeur sur un mot cette fois, on retourne la valeur se trouvant à l'adresse obtenue.

2.3.3 Mutation

Les mutations de vecteurs sont aussi systématiquement générées *inline* pour des raisons de performances. La procédure est essentiellement identique à celle suivie pour *VECTORREF*. *VECTORSET* (figure 2.7) commence par faire un *VECTORREF* sur le

vecteur *x* à l'index *y*, puis fait une assignation de la valeur *z* dans la cellule retournée.

Figure 2.7 – Code généré pour *##vector-set!*

```
#define ___VECTORSET(x,y,z)*(___WORD*)(((___WORD) \
        ___BODY_AS(x,___tSUBTYPED))+((y)<<(___LWS-___TB)))=z;
```

2.3.4 Fonctions utilitaires

Des fonctions utilitaires sont fournies pour des opérations sur les vecteurs nécessitant l'accès aux informations de bas niveau. Celles-ci sont toutes générées *inline*.

Figure 2.8 – Code généré pour *##vector?*

```
#define ___TESTHEADERTAG(x,s)(((x)&___SMASK)==((s)<<___HTB))

#define ___TESTSUBTYPETAG(x,s) \
        ___TESTHEADERTAG(___HEADER(x),s)

#define ___TESTSUBTYPE(x,s)(___TYP((___temp=(x)))==
        ___tSUBTYPED&& \
        ___TESTSUBTYPETAG(___temp,(s)))

#define ___VECTORP(x)___TESTSUBTYPE(x,___sVECTOR)
```

##vector? détermine si un objet Scheme est un vecteur en testant si son type est celui d'un objet sous-typé et si son sous-type est un vector en utilisant un masque sur l'entête. Il est compilé vers l'instruction C *VECTORP* définie dans la figure 2.8.

La fonction *##vector-length*, qui est compilée vers *VECTORLENGTH* (figure 2.9), extrait le champ d'entête de son argument puis utilise un masque pour récupérer le nombre de champs. Le résultat est ensuite décalé de deux bits vers la gauche pour le transformer en *fixnum* (l'étiquette de ceux-ci étant 00).

Figure 2.9 – Code généré pour *##vector-length*

```
#define ___FIX(x)(___CAST(___WORD,x)<<___TB)
#define ___UNTAG_AS(obj,tag)___CAST(___WORD*,(obj)-(tag))
#define ___HEADER(x)(*___UNTAG_AS(x,___tSUBTYPED))
#define ___HD_FIELDS(head)(___CAST(unsigned ___WORD,head)>> \
        (___LF+___LWS))
#define ___VECTORLENGTH(x)___FIX(___HD_FIELDS(___HEADER(x)))
```

2.4 Types

Gambit supporte, comme la plupart des dialectes de Scheme, des enregistrements analogues aux *struct* C. Ceux-ci sont constitués d'un vecteur d'objets Scheme débutant par un champ contenant une référence à un descripteur de type (lui même un enregistrement) suivi d'un ensemble contigu de champs nommés. Le programmeur fournit un appel à une macro de définition qui est ensuite convertie en fonctions spécialisées d'allocation, d'accès, de mutation. Un descripteur de type est aussi créé lors de l'expansion de la macro.

Contrairement à C, ces enregistrements supportent aussi la notion d'héritage. Une définition peut contenir une option permettant de générer une nouvelle macro de définition de type générant des enregistrements ayant le type défini comme parent. Un enregistrement ainsi défini aura aussi le type du parent et tous ses champs.

2.4.1 Définition

Figure 2.10 – Exemple de définition d'un type *point*

```
(define-type point x)
```

La définition des types se fait de façon déclarative. On appelle la macro *define-type*

sur les symboles correspondant aux champs souhaités. Minimalement, on doit fournir le nom du type et le nom des champs (figure 2.10). Cet appel définit un enregistrement de type *point* ayant uniquement un champ *x* et va être étendu vers la description se trouvant à la figure 2.11.

Figure 2.11 – Expansion d'un type *point*

```
(begin

  (define ##type-1-point
    ((let () (##declare (extended-bindings)) ##structure)
     ##type-type
     ((let ()
        (##declare (extended-bindings))
                ##make-uninterned-symbol) "##type-1-point")
     'point
     '8
     #f
     '#(x 0 #f)))

  (define (make-point p1)
        (##declare (extended-bindings))
        (##structure ##type-1-point p1))

  (define (point? obj)
    (##declare (extended-bindings))
    (##structure-direct-instance-of? obj
        (let () (##declare (extended-bindings) (not safe))
        (##type-id ##type-1-point))))

  (define (point-x obj)
        ((let () (##declare (extended-bindings))
        ##direct-typed-structure-ref) obj 4 8
            ##type-1-point point-x))

  (define (point-x-set! obj val)
    ((let () (##declare (extended-bindings))
        ##direct-typed-structure-set!)
        obj val 4 8 ##type-1-point point-x-set!)))
```

Celle-ci comportera un enregistrement (*##type-1-point*) servant de descripteur de type, une fonction *make-point* qui construit un enregistrement de type *point*, une fonction *point ?* qui compare le type d'un objet avec le descripteur de type *##type-1-point*, une fonction *point-x* qui retourne le champ *x* d'un point passé en argument et une fonction

point-x-set ! qui mute le champ *x* d'un point.

En plus des informations strictement nécessaires, il est possible de passer un certain nombre d'options sur le type, entre autres :

- *id :* Donne un identifiant unique au type.
- *extender :* Génère une macro d'héritage.
- *macros :* Génère les définitions sous forme de macros.
- *type-exhibitor :* Fonction sans argument retournant le descripteur de type.

Les champs ont aussi un ensemble d'attributs individuels :

- *read-only :* Ne génère pas de fonction de mutation pour ce champ.
- *equality-skip :* Ne considère pas ce champ pour déterminer l'égalité.
- *unprintable :* N'imprime pas ce champ lorsque l'instance est imprimée.
- *init : cst* Initialise ce champ à la valeur *cst*.

2.4.2 Héritage

L'option *extender :* sert à spécifier que l'expansion de la macro doit générer une macro d'expansion de type ayant le type que l'on spécifie comme parent.

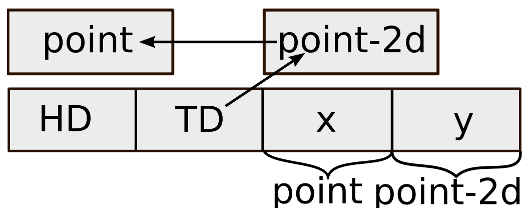
Figure 2.12 – Exemple de définition d'un *point* comme type parent

```
(define-type point extender: type-point x)
(type-point point2d y)
```

Les objets de type *point2d* (figure 2.12) auront la forme donnée à la figure 2.13.

Un champ pour le descripteur de type, le champ *point-x* suivi du champ *point2d-y*. Les accesseurs et mutateurs sont ceux définis par le *define-type* dans lequel le champ est

Figure 2.13 – Représentation d'un point 2d



défini. Les tests de typages nécessaires sont faits de façon récursive sur le descripteur de type, celui-ci comportant un champ pour le type parent. Les accès sans vérification de typage sont identiques aux types sans héritages.

2.4.3 Descripteur de type

Les descripteurs de types sont constitués d'enregistrements ayant comme type *##type-type* (figure 2.14). Celui-ci est défini comme une primitive.

Figure 2.14 – *##type-type*

```
(define ##type-type
  (let ((type
         '#(#f
            ##type-5
            type
            8
            #f
            #(id 1 #f name 5 #f flags 5 #f
               super 5 #f fields 5 #f))))

    ;OK to mutate constant in Gambit
    (##structure-type-set! type type)
    (##subtype-set! type (macro-subtype-structure))
    type))
```

Comme sa définition est nécessaire à la construction d'un enregistrement, on le construit comme un vecteur (l'apostrophe dénote un littéral et le # indique que la liste qui le suit est un vecteur), on mute une autoréférence dans son champ de type et on change ensuite son sous-type pour en faire un enregistrement.

Les champs d'un type sont, dans l'ordre, une référence à *##type-type*, un identifiant

unique pour le type, un symbole correspondant à son nom, un champ signalant les options qui s'appliquent à tout le type, le type parent (faux s'il n'y en a pas) et un vecteur contenant la description de tous les champs.

Cette description est séparée en trois objets par champs : un symbole correspondant au nom du champ, un entier encodant les attributs applicables et la valeur initiale (#f ou faux par défaut).

Tous les types définis par la macro *define-type* (figure 2.15) comportent exactement ces champs.

2.4.4 Define-type-expand

Figure 2.15 – Définition de *define-type*

```
(define-runtime-macro (define-type . args)
  (##define-type-expand 'define-type #f #f args))
```

L'expansion de la macro *define-type* est composée par un appel à la fonction primitive *##define-type-expand*. Celle-ci prend en arguments le nom de l'appelant (au bénéfice des messages d'erreurs), les super-types statique (*#f* par défaut) et dynamique (idem) et les arguments passés à l'appelant (*define-type* ou un *extender*, entre autres).

La primitive retourne une liste de symboles correspondant au code Scheme de l'expansion du *define-type*. Pour ce faire, une fonction *generate* contenant toute la logique de création du code est passée en continuation à la fonction *##define-type-parser* qui va traverser la liste d'arguments de l'appelant et générer les paramètres nécessaires (ou faux lorsque non applicable) tels que donnés à la figure 2.16.

Les paramètres correspondent pour la plupart directement aux options possibles. L'information est passée en mettant le symbole fourni comme valeur ou *#f* si l'option n'est pas présente. Les options n'attendant pas de paramètres ont la valeur *#t* (vrai) s'ils s'appliquent. Le nom du type est propagé dans la valeur *name*, *fields* est une liste de liste décrivant les champs qui se retrouve dans le descripteur de type et total-fields est le

Figure 2.16 – Paramètres de l'expansion de *define-type*

```
(define (generate
         name
         flags
         id
         extender
         constructor
         constant-constructor
         predicate
         implementer
         type-exhibitor
         prefix
         fields
         total-fields)
         ...
```

nombre de champs.

2.4.4.1 ##define-type-parser

La fonction *##define-type-parser* parcourt les arguments et compare les symboles rencontrés avec une table d'association pour les attributs des champs ou une série de conditionnel pour les options globales (les options doivent apparaître dans un ordre spécifique). Si le symbole rencontré est à la bonne position, un traitement est effectué en fonction de sa nature pour composer chacune des variables nécessaires à *generate*.

Une fois toute la liste d'arguments traitée, la continuation (*generate*) est appelée sur l'ensemble de variables. Celle-ci utilise un groupe de fonctions auxiliaires pour construire le code à retourner.

2.4.4.2 generate-fields

La fonction *generate-fields* (figure 2.17) prend la liste de descripteurs des champs de l'objet généré par le *parser* et construit le vecteur de descripteurs qui va se retrouver dans le descripteur de type.

Figure 2.17 – Fonction générant les champs

```
(define (generate-fields)
  (let loop ((lst1 (##reverse fields)) (lst2 '()))
    (if (##pair? lst1)
        (let* ((field (##car lst1))
                (descr (##cdr field))
                (field-name (##vector-ref descr 0))
                (options (##vector-ref descr 4))
                (attributes (##vector-ref descr 5))
                (init
                 (cond ((##assq 'init: attributes)
                        =>
                        (lambda (x)
                         (##constant-expression-value
                             (##cdr x))))
                       (else
                        #f))))
          (loop (##cdr lst1)
                (##cons field-name
                        (##cons options
                                (##cons init
                                        lst2)))))
        (##list->vector lst2))))
```

2.4.4.3 generate-parameters

La fonction *all-fields->rev-field-alist* prend la liste de tous les champs et construit une liste de vecteurs contenant l'information nécessaire à la génération des valeurs initiales. Elle crée aussi un symbole pour chacun des paramètres associés au champ.

Cette liste est passée à la fonction *generate-parameters* qui va construire la liste des paramètres fournis entre autres au constructeur *make-<nom>*. La fonction boucle simplement sur chacun des vecteurs de *rev-field-alist* et ajoute les symboles qui n'ont pas de valeur d'initialisation à la liste des paramètres à passer aux fonctions de construction.

2.4.4.4 generate-initializations

La fonction *generate-initializations* (figure 2.18) parcourt les champs et crée la liste des paramètres qui seront utilisés dans les constructeurs en arguments de l'appel à *##structure*. Si le champ en question a une valeur initiale, il la transforme en littéral, sinon

Figure 2.18 – *generate-initializations*

```
(define (generate-initializations
         field-alist parameters in-macro?)
  (##map (lambda (x)
           (let* ((field-index (##vector-ref (##cdr x) 0))
                  (options (##vector-ref (##cdr x) 1))
                  (val (##vector-ref (##cdr x) 2))
                  (parameter (##vector-ref (##cdr x) 3)))
             (if (##memq parameter parameters)
                 parameter
                 (make-quote
                   (if in-macro?
                       (make-quote val)
                       val)))))
         field-alist))
```

il met le symbole du paramètre. On obtient donc un appel ayant la forme décrite à la figure 2.19.

Figure 2.19 – Expansion du constructeur avec une valeur initiale

```
(define-type foo a (b init: 10) c)
    =>
(define (make-foo p1 p2)
    (##structure type p1 `10 p2))
```

2.4.4.5 generate-getters-and-setters

La fonction *generate-getters-and-setters* boucle sur tous les champs et génère la définition d'une fonction ou une macro d'accès et de mutation si nécessaire. Ces définitions auront la forme donnée aux figures 2.20 et 2.21 pour les accesseurs et les mutateurs respectivement.

La version fonction retourne simplement un littéral du code de l'expansion, alors que la version macro retourne un littéral d'une définition de macro retournant une liste de symboles qui seront substitués au site d'appel.

Figure 2.20 – Construction de la fonction d'accès

```
`((define (,getter-name obj)
    ((let ()
       (##declare (extended-bindings))
       ,getter-method)
      obj
      ,field-index
      ,type-expression
      ,getter-name)))
```

Figure 2.21 – Construction de la macro d'accès

```
`((##define-macro (,setter-name obj val)
    (##list '(let ()
               (##declare (extended-bindings))
               ,setter-method)
             obj
             val
             ,field-index
             ',type-expression
             #f)))
```

2.4.4.6 generate-structure-type-definition

Cette fonction construit la définition de l'expression de type dynamique en encapsulant par valeur une version du type statique dans un define.

2.4.5 Test de typage

L'interface pour les enregistrements fournit des fonctionnalités de test de typage au travers des fonctions primitives *##structure-instance-of ?* et *##structure-direct-instance-of ?*. Chaque *define-type* va définir une fonction <*nom*> *?* (figure 2.22) qui fait appel à l'une de ces primitives (direct si l'enregistrement n'est pas d'un type défini par héritage) en construisant une expression de la forme suivante :

Les fonctions de test de typage testent si l'objet *obj* est un enregistrement et si c'est le cas, compare son *id* avec celui passé en paramètre. La version directe va retourner

Figure 2.22 – Expansion de tests de type

```
(define (point? obj)
 (##structure-direct-instance-of? obj <type-id of point>))

(define (point2d? obj)
 (##structure-instance-of? obj <type-id of point2d>))
```

le résultat de cette comparaison directement alors que la version prenant en compte l'héritage va monter récursivement dans l'arbre d'héritage tant qu'il existe des super-type ou qu'elle n'a pas rencontré un type correspondant.

2.4.6 Construction

La construction d'enregistrements se fait par l'entremise de la fonction primitive *##structure*. Un appel à cette fonction est construit par *define-type* dans la définition de *make-<nom>* (figure 2.23), un constructeur spécialisé pour chaque type défini.

Figure 2.23 – Expansion du constructeur

```
(define (make-point2d p1 p2)
 (##structure <type point2d> p1 p2))
```

Cette version spécialisée permet de traiter aisément les paramètres avec valeurs initiales et la propagation du descripteur de type sans la nécessité d'avoir un *type-exhibitor* explicite.

La fonction *##structure* (figure 2.24) s'appelle récursivement en comptant le nombre de champs passé en paramètres (le compte débute à 1 pour prendre en compte le champ du type). Une fois toute la liste traversée, le dernier appel construit un vecteur de cette taille dont tous les champs sont initialisés au descripteur de type. Le sous-type de ce vecteur est ensuite changé pour celui d'un enregistrement et le vecteur est retourné à l'appelant. Chacun des appels récursifs va ensuite muter la cellule correspondant au champ passé en

Figure 2.24 – Primitive *##structure*

```
(define-prim (##structure type . fields)

  (define (make-struct fields i)
    (if (##pair? fields)
      (let ((s (make-struct (##cdr fields) (##fx+ i 1))))
        (##unchecked-structure-set!
             s (##car fields) i type #f)
        s)
      (let ((s (##make-vector i type)))
        (##subtype-set! s (macro-subtype-structure))
        s)))

  (make-struct fields 1))
```

paramètre à la valeur fournie. Le vecteur est ensuite retourné au niveau précédent jusqu'à ce que tous les champs soient traités. L'enregistrement est la valeur de retour finale.

Pour des raisons de performances, tous les appels non *safe* à *##structure*, c.a.d. dans des blocs de code portant cette déclaration ce qui permet au compilateur d'éliminer les tests de typages, sont construits sous forme *inline*. Les appels sont traduits par la fonction *targ-apply-vector* et suivent exactement la même procédure que l'allocation *inline* d'un vecteur. La seule différence est l'utilisation en paramètre *kind* du symbole *'structure* qui fait que le bloc de code créant l'entête met une valeur séciale(*sSTRUCTURE*) comme sous type plutôt qu'un sous-type de vecteur.

2.4.7 Références et mutations

Les références sont implémentées d'une façon similaire aux allocations, soit quelques fonctions primitives et du sucre syntaxique généré par *define-type*. Comme le paramètre passé à la fonction peut être un objet Scheme quelconque, lorsque le code est exécuté en mode *safe*, il est nécessaire d'ajouter un test de typage (figure 2.25).

On l'a vu précédemment, ceux-ci diffèrent selon que le type est hérité ou non donc les fonctions primitives incluent une version faisant un test de type direct, une avec test de type récursif et, finalement, une version qui ne fait pas de vérification et qui peut être

Figure 2.25 – Primitive *##structure-ref*

```
(define-prim (##structure-ref obj i type proc)
  (if (##structure-instance-of? obj (##type-id type))
    (##unchecked-structure-ref obj i type proc)
    (##raise-type-exception
     1
     type
     (if proc proc ##structure-ref)
     (if proc (##list obj) (##list obj i type proc)))))
```

étendue *inline*. La version non typée contient aussi la logique d'accès utilisée par les deux autres. La macro *define-type* génère toujours des versions typées, mais celles-ci peuvent être spécialisées par le compilateur dans une version sans vérification lorsque c'est possible.

Figure 2.26 – Extension *inline* de *##unchecked-structure-ref*

```
(targ-op "##unchecked-structure-ref"
         (targ-ifjump-apply-u "UNCHECKEDSTRUCTUREREF"))
    =>
#define ___UNCHECKEDSTRUCTUREREF(x,y,type,proc)
          ___VECTORREF(x,y)
```

La version *inline* (figure 2.26) est traduite par un simple appel à la macro C *VECTORREF* en ignorant certains des paramètres passés à *##unchecked-structure-ref* afin que l'interface soit toujours constante entre les trois versions pour permettre les spécialisations.

Les mutations sont traitées de façon identique aux références, mis à part le fait que la cellule référée est modifiée et que la valeur retournée est la constante primitive *##void*. Les opérations de spécialisation et d'*inlining* sont faites de la même manière que les références et l'accès sans test de typage est ultimement fait avec un appel à *VECTORSET*.

CHAPITRE 3

ENREGISTREMENTS CONTENANT DES CHAMPS TYPÉS STATIQUEMENT

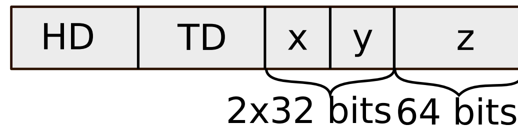
De nombreuses applications utilisant de grands ensembles de structures les utilisent pour contenir des données numériques ayant un lien logique entre elles, comme par exemple des points dans un espace donc chaque champ correspond à sa coordonnée dans une des dimensions. En général, ces données numériques doivent être contenues dans un espace correspondant à un multiple entier de la taille des objets Scheme. Par exemple, un point bidimensionnel va utiliser deux mots, un pour chacun des champs, en plus de la mémoire supplémentaire utilisée par la représentation des objets en mémoire. Pire, si les coordonnées sont des nombres à virgules flottantes, l'objet devra en général contenir une référence vers d'autres objets alloués sur le tas. Il est toutefois possible en utilisant l'information de typage pour les départager, d'utiliser des types numériques à taille fixe dans une structure ce qui permet un bien meilleur contrôle sur la taille des données allouées.

3.1 Survol

Pour économiser de l'espace sur les champs des enregistrements contenant des données numériques à précision fixe, on ajoute la possibilité de mettre des annotations de types aux champs comme paramètres optionnels. Ces annotations vont entraîner la définition de méthodes d'accès et de mutation spécialisées pour ce type. Évidemment, l'ajout de champs numériques de taille hétérogène dans les enregistrements doit être pris en compte par le *garbage collector*. On modifie le descripteur de type en lui adjoignant des données supplémentaires permettant au collecteur de parcourir les enregistrements partageant ce type.

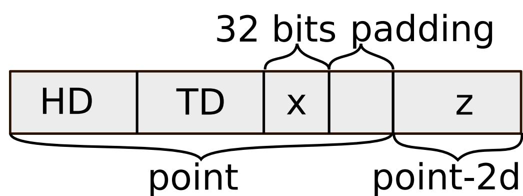
L'inclusion de champs ayant des tailles fixes, mais hétérogènes (figure 3.1) (par opposition, par exemple, à des vecteurs d'entiers 32 bits, qui ont tous la même taille et un contenu du même type et peuvent donc être gérés de façon identique) et potentiellement

Figure 3.1 – Enregistrements avec des champs de tailles hétérogènes



mélangés avec des champs Scheme normaux fait en sorte que la position relative des champs diffère potentiellement en fonction de la taille des mots machine. Comme le code C généré par le compilateur doit être compatible tant en 32 qu'en 64 bits, il devient donc nécessaire de propager les positions précises de chaque champ aux fonctions qui les nécessitent et d'indexer les enregistrements avec une granularité d'un octet près.

Figure 3.2 – Enregistrements multiniveaux avec *padding*



Pour limiter la consommation de mémoire (et donc la pression sur le *garbage collector*), on souhaite que ces données aient la représentation la plus compacte possible, mais pour des raisons de performances il est souhaitable d'ajouter des contraintes d'alignement. On aligne donc les enregistrements sur les frontières des niveaux d'héritage en ajoutant du *padding* (figure 3.2) pour s'assurer que chaque frontière d'héritage se retrouve sur un multiple de la taille des mots.

On doit donc modifier la macro *define-type*, les descripteurs de types, introduire de nouvelles primitives d'accès et de mutations et modifier le *garbage collector* pour prendre en compte ces changements.

3.2 Modifications à *define-type*

Les changements nécessaires à l'ajout d'enregistrements typés commencent avec l'ajout de paramètres indiquant le type d'un champ aux définitions d'enregistrements (figure 3.3).

Figure 3.3 – Exemple de définition d'un point typé

```
(define-type point-8bit (x int8:))
```

On ajoute simplement un ensemble d'attributs à la table associative existante détaillant chacun des types que l'on souhaite supporter. Ceux-ci sont dans notre cas : *int8, int16, int32, int64, uint8, uint16, uint32, uint64, float32 et float64* soit respectivement des entiers signé et non signé de 8, 16, 32 et 64 bits et des nombres à virgule flottante de 32 et 64 bits. On assigne à chacun un entier multiple de 16 (parce que les valeurs plus basses sont déjà utilisées) entre 16 et 160 afin de pouvoir utiliser un simple masque (240) et un décalage à droite pour l'extraire.

Figure 3.4 – Primitive *##sizeof-typed-field*

```
(define-prim  (##sizeof-typed-field opt)
  (let* ((table '#(1 2 4 8 1 2 4 8 4 8))
         (i (##bitwise-and opt 240))
         (index (##fixnum.arithmetic-shift-right i 4)))
    (if (and (##fixnum.> index 0)
             (##fixnum.< index 11))
        (##vector-ref table (##fixnum.- index 1))
        #f)))
```

Une primitive extrayant la taille d'un champ en fonction de sa valeur d'attribut (figure 3.4) est aussi ajoutée, car il s'agit d'un calcul qui est effectué fréquemment. On extrait simplement l'entier référent à la position de l'attribut dans la liste et on indexe un vecteur contenant les tailles si l'index correspond à un champ typé. On retourne faux (*#f*) sinon pour faciliter les tests faisant la distinction entre ceux-ci et les champs normaux.

Des changements substantiels sont aussi nécessaires à la logique de *##define-type-expand*. Comme on permet de définir des champs de tailles variables à l'intérieur d'un enregistrement, il est nécessaire de calculer le décalage requis pour chaque champ à partir du descripteur de type. On doit obtenir ces informations en 32 et 64 bits, car il s'agit des deux formats de mots machines supportés par le compilateur. Comme d'autres

informations calculées à partir de l'ensemble des champs seront aussi nécessaires, on calcule le tout en une passe à l'aide d'une fonction auxiliaire (définie dans un bloc *let*).

Cette fonction (figure 3.5) commence par définir d'autres fonctions auxiliaires, soit *calc* qui calcule la position de fin d'un champ en prenant sa taille (obtenue avec la fonction *##sizeof-typed-field*), la taille d'un mot et la position de la fin du mot précédent. Les fonctions *cpd2* (*calculate-padding-2-args*) et *pad* servent au calcul du rembourrage nécessaire entre deux niveaux d'héritage pour aligner les champs sur un multiple de *i*.

Le corps de la fonction est formé de la boucle *loop* qui prend en arguments l'indice dans le vecteur de description des champs qui commencent à 1, car on s'intéresse au marqueur d'attributs. Rappelons que le vecteur de descriptions comporte 3 valeurs pour chaque champ soit son nom, le marqueur d'attributs et la valeur initiale.

Les arguments *sc* et *tc* correspondent au nombre d'objets Scheme et de bits de champs typés respectivement rencontrés dans les champs précédents. Ils serviront au calcul de la taille maximale en mots de l'enregistrement. Viennent ensuite les arguments *o32*, *o64*, *p32* et *p64*. Ceux-ci correspondent aux valeurs courantes de l'information que l'on calcule sur la position de chacun des champs. Les valeurs de *o32* et *o64* correspondent au décalage en bits depuis le pointeur vers l'enregistrement et sont stockées dans une liste. La valeur initiale correspond à la fin du descripteur de type en 32 et 64 bits respectivement. Les valeurs *p32* et *p64* accumulent dans une liste la position des champs autres que le descripteur de type qui doivent être pris en compte par le *garbage collector*. On omet le descripteur de type puisque cette information y sera stockée et qu'on doit donc le traverser pour y accéder.

Les derniers arguments, *typed ?*, *b* et *typ* sont respectivement un indicateur que l'enregistrement contient au moins un champ typé, une liste des indices auquel on passe d'un type parent à son héritier et une liste des marqueurs de types de chaque champ rencontré précédemment.

La boucle itère sur la concaténation des vecteurs de description des champs de chaque niveau d'héritage par bond de 3 (le nombre d'attributs associés à chaque champ). À chaque itération, on calcule les nouvelles valeurs de position en fonction du type du champ et on accumule les valeurs dans les paramètres de l'appel suivant en prenant en

Figure 3.5 – Calcul des paramètres requis pour les champs typés

```
(stcount
 (let* ((len (##vector-length all-type-fields))
        (calc (lambda (s i o) (+ o (if s s i))))
        (cpd2 (lambda (v i)
                (let ((m (modulo v i)))
                  (if (= m 0)
                      0
                      (- i m))))))
        (pad (lambda (s o32 o64)
               (let* ((v32 (calc s 4 (car o32)))
                      (v64 (calc s 8 (car o64)))
                      (p32 (cpd2 v32 4))
                      (p64 (cpd2 v64 8)))
                 (list (+ p32 v32) (+ p64 v64))))))
   (let loop ((i 1)
              (sc 0)
              (tc 0)
              (o32 (##list 4))
              (o64 (##list 8))
              (p32 '())
              (p64 '())
              (typed? #f)
              (b breakpoints)
              (typ '()))
     (if (>= i len)
         (##vector sc tc o32 o64 p32 p64 typed? typ)
         (let* ((tag (##vector-ref all-type-fields i))
                (s (##sizeof-typed-field tag))
                (ntag (##bitwise-and tag 240))
                (nv (if (= i (- (car b) 2))
                        (cons (cdr b) (pad s o32 o64))
                        (list b (calc s 4 (car o32))
                                (calc s 8 (car o64)))))
                (nb (car nv))
                (n32 (cons (cadr nv) o32))
                (n64 (cons (caddr nv) o64)))
           (if s
               (loop (+ i 3) sc (+ tc s)
                       n32 n64 p32 p64 #t nb (cons ntag typ))
               (loop (+ i 3) (+ sc 1) tc n32 n64
                     (cons (car o32) p32)
                     (cons (car o64) p64)
                    typed? nb (cons ntag typ))))))))
```

compte le rembourrage nécessaire. Lorsqu'on a parcouru tout le vecteur, on retourne un vecteur contenant les 8 valeurs requises, soit tous les paramètres sauf les *breakpoints* et l'indice courant.

Figure 3.6 – Extraction des paramètres supplémentaires requis pour les champs typés

```
(scount (##vector-ref stcount 0))

(tcount (##vector-ref stcount 1))

(pos32 (##vector-ref stcount 2))

(pos64 (##vector-ref stcount 3))

(spos32 (##list->vector
          (##reverse (##vector-ref stcount 4))))

(spos64 (##list->vector
          (##reverse (##vector-ref stcount 5))))

(typed (##vector-ref stcount 6))

(max-words (let* ((y (car pos32))
                  (x (##fixnum.quotient y 4)))
             (if (= 0 (modulo y 4)) x (+ x 1))))

(type-tags (##reverse (##vector-ref stcount 7)))
```

Les champs sont ensuite extraits (figure 3.6) dans des variables distinctes avec le traitement nécessaire pour les utiliser. Le nombre de mots maximal est toujours le nombre de mots complets en 32 bits. On le calcule en prenant le quotient de la position de fin du dernier champ de l'enregistrement et de la taille d'un mot (4 octets). Si celui-ci n'est pas un entier, on arrondit à la valeur entière suivante. Certaines des listes sont renversées, car les listes générées dans *stcount* sont en ordre inverse (du dernier champ au premier) afin de faciliter l'extraction d'information, comme la position de fin du dernier élément de l'enregistrement.

On utilise ensuite l'information calculée précédemment pour étendre la définition des descripteurs de types statique (figure 3.7) et dynamique. Ces informations servent exclusivement au *garbage collector*. Le champ *max-words* ici compte seulement les

Figure 3.7 – Définition du descripteur de type statique

```
(type-static
 (##structure
  ##type-type
  (if generative?
      (##make-uninterned-symbol augmented-id-str)
      (##string->symbol augmented-id-str))
  name
  flags
  super-type-static
  type-fields
  (##fixnum.- max-words 1)
  tcount
  spos32
  spos64))
```

champs définis par l'usager et ne compte pas le mot utilisé pour le descripteur de type.

Figure 3.8 – Expansion des mutateurs et des accesseurs dans *##define-type-expand*

```
,@(let loop ((lst1 (##reverse fields))
             (lst2 '())
             (p32 (cdr pos32))
             (p64 (cdr pos64)))
    (if (##pair? lst1)
        (loop (##cdr lst1)
              (generate-getter-and-setter
                   (##car lst1) lst2 p32 p64)
              (cdr p32)
              (cdr p64))
        lst2))
```

La génération des accesseurs et mutateurs (figure 3.8) doit aussi être modifiée pour prendre en compte la dépendance entre l'index d'un champ et les précédents. On étend donc la boucle qui construit la liste des définitions des fonctions d'accès et de mutation pour consommer la liste des décalages calculés précédemment. On saute le premier élément de chacune des listes, car il s'agit du descripteur de type.

La fonction commence en extrayant (figure 3.9) la valeur de l'index du champ dont on veut générer les méthodes. La construction de l'appel à la méthode (figure 3.10)

Figure 3.9 – Construction des paramètres des champs

```
(opt-code
 (##vector-ref descr 4))

(table-index (##fixnum.arithmetic-shift-right
 (##bitwise-and opt-code 240) 4))

(field-index-32 (##car pos32))

(field-index-64 (##car pos64))
```

se fait en deux étapes. Si le champ comporte un accesseur, on construit le symbole correspondant à son nom, puis le code de la méthode à appeler. Pour cette dernière, on choisit la primitive d'accès adéquate en fonction du marqueur d'attribut *opt-code* et du paramètre de *##define-type-expand extender* de façon analogue à *##sizeof-typed-field*. Ensuite, selon qu'il s'agit d'une définition sous forme de macro ou de fonction, on génère le code approprié. Les primitives d'accès originales sont ici complètement remplacées par de nouvelles primitives spécialisées au type de données attendu et prenant un paramètre de plus pour pouvoir gérer la différence de taille au niveau du *runtime*. L'utilisation des fonctions spécialisées est identique à celles générées pour des enregistrements non typés (figure 3.11).

La construction des fonctions de mutations se fait de la même façon, mis à part un paramètre *val* correspondant à la valeur qui sera inscrite dans le champ ajouté aux deux définitions générées. Ces fonctions auront donc la même forme que n'importe quel accesseur.

La construction du constructeur est modifiée de façon similaire (figure 3.12). Un test les différencie et appelle une version distincte de la primitive d'allocation *##typed-structure* (figure 3.13). Celle-ci a besoin d'informations supplémentaires pour générer l'allocation et l'initialisation des champs. Ces informations lui sont donc passées en paramètre en plus des arguments standards.

Figure 3.10 – Construction des appels aux accesseurs

```
(getter-def
 (if getter
     (let ((getter-name
            (if (##eq? getter #t)
                (##symbol-append prefix
                                 name  '- field-name)
                getter))

           (getter-method
            (let ((table
                   (##vector '##int8-structure-ref
                             '##int16-structure-ref
                             ...
                             '##float64-structure-ref))
                  (direct-table
                   (##vector '##direct-int8-structure-ref
                             '##direct-int16-structure-ref
                             ...
                             '##direct-float64-structure-ref)))
              (if extender
                  (if (and (> table-index 0)
                           (<= table-index 10))
                      (##vector-ref table (##fixnum.- table-index 1))
                      '##typed-structure-ref)
                  (if (and (> table-index 0)
                              (<= table-index 10))
                      (##vector-ref direct-table
                        (##fixnum.- table-index 1))
                      '##direct-typed-structure-ref)))))

       (if macros?
           `((##define-macro (,getter-name obj)
               (##list '(let ()
                           (##declare (extended-bindings))
                           ,getter-method)
                       obj
                       ,field-index-32
                       ,field-index-64
                       ',type-expression
                       #f)))

           `((define (,getter-name obj)
               ((let ()
                  (##declare (extended-bindings))
                  ,getter-method)
                obj
                ,field-index-32
                ,field-index-64
                ,type-expression
                ,getter-name)))))
     `()))
```

Figure 3.11 – Construction d'un point 1d 8 bits

```
(define p (make-point-8bit 11))
(point-8bit-x p) ;; => 11
```

3.3 Primitive d'allocation

La primitive *##typed-structure* sert à construire les enregistrements typés. Même si ces informations sont disponibles dans le descripteur de type, on fournit en paramètres le nombre de mots maximal, les attributs correspondant à chacun des champs et les décalages en 32 et 64 bits afin de permettre la construction de version *inline* de la primitive. En effet, le descripteur de type n'est pas nécessairement une valeur constante au moment de la compilation, ce qui empêche les valeurs nécessaires d'être calculées durant l'extension *inline* de la primitive. Pour contourner ce problème, on fournit en argument des littéraux contenant toute l'information requise. Il est donc inutile de la recréer dans *##typed-structure*.

L'implantation se fait de manière analogue à celle de *##structure*. On alloue un vecteur ayant un nombre de champs de type *scmobj* égal au nombre maximum de mots utilisé par l'enregistrement puis on initialise chacun des champs en consommant le code d'attribut et la position en 32 et 64 bits de chacun des champs. On change ensuite le sous-type du vecteur pour en faire un enregistrement.

3.4 Primitives d'accès et de mutations

Les primitives sans test de typage (figure 3.14) sont chacune spécialisées à un type donné et sont construites avec un simple appel vers un bloc de code C à travers l'interface vers C.

L'implémentation de ces macros (figure 3.15) est largement semblable à celle de *VECTORREF* et *VECTORSET*. On calcule l'adresse à laquelle on doit référer en extrayant la position du début des champs avec *BODY* ce qui fait un calcule avec un masque pour

Figure 3.12 – Construction de l'appel au constructeur *make-<type>*

```
,@(if constructor
      (let ((constructor-name
             (if (##pair? constructor)
                 (##car constructor)
                 constructor)))

        (if typed
            (if macros?
                `((##define-macro (,constructor-name ,@parameters)
                    (##list '(let ()
                               (##declare (extended-bindings))
                               ##typed-structure)
                            ',type-expression
                            ',max-words
                            '',type-tags
                            '',(##reverse pos32)
                            '',(##reverse pos64)
                            ,@(generate-initializations
                            field-alist
                            parameters
                            #t))))
                `((define (,constructor-name ,@parameters)
                    (##declare (extended-bindings))
                    (##typed-structure
                     ,type-expression
                     ,max-words
                     ',type-tags
                     ',(##reverse pos32)
                     ',(##reverse pos64)
                     ,@(generate-initializations
                     field-alist
                     parameters
                     #f)))))
            (if macros?
                `((##define-macro (,constructor-name ,@parameters)
                    (##list '(let ()
                               (##declare (extended-bindings))
                               ##structure)
                            ',type-expression
                            ,@(generate-initializations
                            field-alist
                            parameters
                            #t))))
                `((define (,constructor-name ,@parameters)
                    (##declare (extended-bindings))
                    (##structure
                     ,type-expression
                     ,@(generate-initializations
                     field-alist
                     parameters
                     #f)))))))
      '())
```

Figure 3.13 – Primitive *##typed-structure*

```
(define-prim (##typed-structure
    type mw typ pos32 pos64 . fields)

 (define (make-struct-typed fields)

    (let ((v (##make-vector mw type)))

      (define (init fields pos32 pos64 types)

        (if (##pair? fields)
          (begin

            ((##struct-set!-by-code (##car types))
                                    v (##car fields)
                (##car pos32) (##car pos64) type #f)

            (init (##cdr fields)
                (##cdr pos32) (##cdr pos64) (cdr types)))

          (begin (##subtype-set! v
                                (macro-subtype-structure))
                     v)))

      (init fields pos32 pos64 typ)))

  (make-struct-typed fields))
```

éliminer des bits de marquages arbitraires. On ajoute ensuite le décalage approprié à la taille des mots machine en transformant la valeur marquée passé en paramètre avec une opération de *shift* droite. La valeur retournée va ensuite être transformée en objet Scheme dans le cas d'une référence ou convertie en entier C puis stockée à l'adresse et avec le type approprié à l'aide d'un *casting* pour la mutation. On s'assure ainsi de ne toucher qu'aux bits nécessaires lors des manipulations.

Évidemment, les conversions vers des entiers de 32 et 64 bits sont passablement plus complexes (figure 3.16), car elles doivent prendre en compte la possibilité de débordement et l'allocation sur ce qui peut être un objet Scheme de plusieurs mots.

Comme les appels de primitive qui sont construits par *##define-type-expand* utilisent

Figure 3.14 – Primitive *##unchecked-int8-structure-ref* et *set !*

```
(define-prim (##unchecked-int8-structure-ref
       obj ofs32 ofs64 type proc)

  (let* ((v (##c-code
            "___RESULT = ___INT8STRUCTREF(___ARG1, ___ARG2,
                ___ARG3, ___ARG4, ___ARG5);"
            obj ofs32 ofs64 type proc)))
    v))

(define-prim (##unchecked-int8-structure-set!
       obj val ofs32 ofs64 type proc)

  (let* ((v (##c-code
             "___INT8STRUCTSET(___ARG1, ___ARG2, ___ARG3,
                ___ARG4, ___ARG5, ___ARG6);
              ___RESULT = ___VOID;"
             obj val ofs32 ofs64 type proc)))
    v))
```

toujours des tests de types, mais que ceux-ci doivent être éliminés lorsque le code est compilé en mode *not safe*, on ajoute pour chaque type des spécialisations (figure 3.17) permettant de convertir les appels typés en appels *unchecked*.

Les fonctions primitives (figure 3.18) qui vont servir d'interface avec la macro *define-type* sont elles aussi spécialisées à chacun des types supportés et fonctionnent de la même façon que *##structure-ref* et *##structure-set !*, mais en utilisant les versions typées des accès sans test de typage et ont donc deux paramètres d'index.

3.5 Optimisation *inline*

Pour des raisons de performances, le code compilé en mode *not safe* est converti en une séquence d'opérations C. Les versions avec tests de typages des accesseurs et mutateurs sont spécialisées vers la version *unchecked* correspondante puis celle-ci est compilée comme un appel direct à la macro C adéquate. Aucun traitement particulier n'est nécessaire.

L'extension *inline* de l'allocateur se fait par le biais d'une fonction spécialisée (figure 3.19). Celle-ci émet le code en utilisant les paramètres constants qui lui sont fournis par

Figure 3.15 – Code généré pour les accesseurs et mutateurs

```
#if ___WS == 4

#define ___INT8STRUCTREF(x,y32,y64,type,proc) \
 ___FIX(*((___S8*) (((___WORD)___BODY(x))+ \
 ((y32)>>___TB))))

#define ___INT8STRUCTSET(x,z,y32,y64,type,proc)  \
 *((___S8*) (((___WORD)___BODY(x))+ \
 ((y32)>>___TB))) = ___INT(z);

#else

#define ___INT8STRUCTREF(x,y32,y64,type,proc) \
 ___FIX(*((___S8*) (((___WORD)___BODY(x))+ \
 ((y64)>>___TB))))

#define ___INT8STRUCTSET(x,z,y32,y64,type,proc) \
 *((___S8*) (((___WORD)___BODY(x))+ \
 ((y64)>>___TB))) = ___INT(z);

#endif
```

define-type.

Cette fonction (figure 3.20) extrait de la liste des arguments de l'appel qui est compilé une variable correspondant au descripteur de type (*t*), une valeur constante entière *mw* donnant le nombre maximal de mots, des listes littérales pour les codes d'attributs et la position des champs en 32 et 64 bits. De plus, le nombre d'arguments et les variables ou constantes dénotant les valeurs associées à chacun des champs sont aussi extraits. Ici, la fonction *obj-val* permet d'extraire une valeur connue au moment de la compilation d'une variable. Une fonction de test *obj ?* permet aussi de déterminer si la valeur d'un champ est connue. Le test n'est pas nécessaire ici, car par construction, les appels sont générés d'une seule source (*define-type*).

Contrairement à l'implémentation de *targ-apply-vector*, on connaît nécessairement, ici, la taille à allouer (le nombre maximal de mots de l'objet) et les macros à utiliser pour initialiser le tas, décaler le pointeur d'allocation et récupérer la valeur de retour. Les seules parties qui diffèrent selon le contenu de l'enregistrement typé sont la valeur des champs,

Figure 3.16 – *Unboxing* des entiers signés de 32 bits et non signés de 64 bits

```
#define ___S32UNBOX(x) \
(___TYP((___temp=x)) == ___tFIXNUM \
 ? ___INT(___temp) \
 : ___BIGAFETCHSIGNED(___BODY_AS(___temp,___tSUBTYPED),0))

#define ___U64UNBOX(x) \
(___TYP((___temp=x)) == ___tFIXNUM \
 ? ___U64_from_UM32(___CAST_U32(___INT(___temp))) \
 : (___HD_BYTES(___HEADER(___temp)) == (1<<2) \
    ? ___U64_from_UM32(___BIGAFETCH( \
        ___BODY_AS(___temp,___tSUBTYPED),0)) \
    : ___U64_from_UM32_UM32(\
        ___BIGAFETCH(___BODY_AS(___temp,___tSUBTYPED),1), \
        ___BIGAFETCH(___BODY_AS(___temp,___tSUBTYPED),0))))
```

Figure 3.17 – Spécialisation des primitives d'accès

```
(def-spec "##int8-structure-ref"
        (spec-u "##unchecked-int8-structure-ref"))

(def-spec "##int8-structure-set!"
        (spec-u "##unchecked-int8-structure-set!"))

(def-spec "##direct-int8-structure-ref"
        (spec-u "##unchecked-int8-structure-ref"))

(def-spec "##direct-int8-structure-set!"
        (spec-u "##unchecked-int8-structure-set!"))
```

leur type et les décalages requis pour les initialiser. On utilise les codes d'attributs de chacun des champs pour différencier le type du champ. On fournit à l'initialisateur les valeurs à passer en paramètre aux macros qui seront émises par le biais des listes en arguments. La position sur la pile des variables qui doivent être passées en arguments à ces macros est construite à l'aide des fonctions standards, soit *targ-opnd* et *targ-opnd-flo* dans le cas des arguments qui sont des nombres à virgule flottante.

Les macros qui sont émises (figure 3.21) fonctionnent de la même façon que pour les enregistrements non typés, mis à part le traitement particulier nécessaire à l'initialisation. Les macros *ADD_<Type>_TSTRUCTURE_ELEM* reçoivent en paramètre des entiers non

Figure 3.18 – Primitives d'accès et de mutations

```
(define-prim (##direct-int8-structure-ref obj ofs32 ofs64 type proc)
  (if (##structure-direct-instance-of? obj (##type-id type))
      (##unchecked-int8-structure-ref obj ofs32 ofs64 type proc)
      (begin
        (##raise-type-exception
         1
         type
         (if proc proc ##direct-int8-structure-ref)
         (if proc (##list obj)
                  (##list obj ofs32 ofs64 type proc))))))

(define-prim (##direct-int8-structure-set!
                        obj val ofs32 ofs64 type proc)
  (if (##structure-direct-instance-of? obj (##type-id type))
      (begin
        (##unchecked-int8-structure-set! obj val ofs32 ofs64 type proc)
        (##void))
      (begin
        (##raise-type-exception
         1
         type
         (if proc proc ##direct-int8-structure-set!)
         (if proc (##list obj val)
                  (##list obj val ofs32 ofs64 type proc))))))

(define-prim (##int8-structure-ref obj ofs32 ofs64 type proc)
  (if (##structure-instance-of? obj (##type-id type))
      (##unchecked-int8-structure-ref obj ofs32 ofs64 type proc)
      (begin
        (##raise-type-exception
         1
         type
         (if proc proc ##int8-structure-ref)
         (if proc (##list obj)
                  (##list obj ofs32 ofs64 type proc))))))

(define-prim (##int8-structure-set!
                        obj val ofs32 ofs64 type proc)
  (if (##structure-instance-of? obj (##type-id type))
      (begin
        (##unchecked-int8-structure-set! obj val ofs32 ofs64 type proc)
        (##void))
      (begin
        (##raise-type-exception
         1
         type
         (if proc proc ##int8-structure-set!)
         (if proc (##list obj val)
                  (##list obj val ofs32 ofs64 type proc))))))
```

Figure 3.19 – Définition de l'expansion *inline* de *##typed-structure*

```
(targ-op "##typed-structure"
         (targ-apply-typed-structure #f))
```

boxés. On interprète le pointeur de tas où l'allocation de l'enregistrement est faite comme un vecteur d'octets et on lui ajoute un décalage de 4 (pour l'entête) plus le décalage calculé dans *##define-type-expand* octets pour obtenir l'adresse à laquelle on va faire l'initialisation. On réinterprète ensuite cette adresse comme référant au type du champ et on assigne la valeur passée en paramètre à l'emplacement en question après l'avoir transformée en valeur C.

Les tests de typages, les références vers les types des enregistrements et leurs mutations sont aussi optimisés de façon similaire par Gambit, mais aucune modification n'est nécessaire pour supporter les champs typés.

3.6 Modifications au *garbage collector*

La présence de champs contenant des valeurs C sans marqueurs de typage à des endroits arbitraires dans les enregistrements typés force la présence d'annotation supplémentaire pour le *garbage collector*. Afin de propager cette information à un endroit accessible durant l'exécution sans ajouter des champs sur chacun des enregistrements, on étend les descripteurs de types avec l'information nécessaire (figure 3.22). On ajoute donc des champs dénotant le nombre d'objets Scheme que fait l'objet (*scount*), le nombre d'octets de champs typés contenus par l'enregistrement (*tcount*) et des vecteurs contenant la position relative au début de l'enregistrement de chacun des champs pouvant contenir une référence (*pos32* et *pos64*). Dans le cas du descripteur de type des types, *##type-type*, tous les champs sont des objets Scheme et son champ contenant la liste des champs contenus est la même que pour tout autre descripteur de type.

Pour différencier les enregistrements, qui peuvent maintenant avoir une distribution irrégulière de champs qui doivent être parcourus, on modifie la fonction *scan* (l'ajout

Figure 3.20 – Code générant *##typed-structure inline*

```
(define (targ-apply-typed-structure proc-safe?)
  (targ-setup-inlinable-proc
    proc-safe?
    #f
    #f
    (lambda (opnds sn)
      (let* ((n    (length opnds))
             (t     (list-ref opnds 0))
             (mw    (obj-val (list-ref opnds 1)))
             (tags  (obj-val (list-ref opnds 2)))
             (pos32 (obj-val (list-ref opnds 3)))
             (pos64 (obj-val (list-ref opnds 4)))
             (rest  (cddr (cdddr opnds))))

          (let ()
            (define (compute-space n) mw)
            (define begin-allocator-name "BEGIN_ALLOC_TSTRUCTURE")
            (define end-allocator-name "END_ALLOC_TSTRUCTURE")
            (define getter-operation "GET_TSTRUCTURE")

            (define (add-element tag i32 i64 elem)
              (case (##fixnum.arithmetic-shift-right tag 4)
                ((1)  (list "ADD_S8_TSTRUCTURE_ELEM"
                            i32 i64 (targ-opnd elem)))
                  ...
                ((8)  (list "ADD_U64_TSTRUCTURE_ELEM"
                            i32 i64 (targ-opnd elem)))
                ((9)  (list "ADD_F32_TSTRUCTURE_ELEM"
                            i32 i64 (targ-opnd-flo elem)))
                ((10) (list "ADD_F64_TSTRUCTURE_ELEM"
                            i32 i64 (targ-opnd-flo elem)))
                (else (list "ADD_TSTRUCTURE_ELEM"
                            i32 i64 (targ-opnd elem)))))

            (targ-heap-reserve-and-check
              (compute-space n)
              (targ-sn-opnds opnds sn))

            (let ()
              (targ-emit (list begin-allocator-name mw))
              (targ-emit (add-element 0 0 0 t)) ;; Type descriptor
              (let loop ((tag tags) (p32 pos32) (p64 pos64) (r rest))
                (if (pair? r)
                    (begin
                      (targ-emit (add-element
                        (car tag) (car p32) (car p64) (car r)))
                      (loop (cdr tag) (cdr p32) (cdr p64) (cdr r)))))
              (targ-emit (list end-allocator-name mw))
              (list getter-operation mw)))))))
```

Figure 3.21 – Macros servant à générer l'expansion de *##typed-structure*

```
#define ___BEGIN_ALLOC_STRUCTURE(n) \
        ___hp[0]=___MAKE_HD_WORDS(n,___sSTRUCTURE);

#define ___ADD_STRUCTURE_ELEM(i,val)___hp[i+1]=(val);

#define ___END_ALLOC_STRUCTURE(n)___ALLOC(n+1);

#define ___GET_STRUCTURE(n)___TAG((___hp-n-1),___tSUBTYPED)

#define ___ADD_U64_TSTRUCTURE_ELEM(i32,i64,val) \
        *((___U64*)  (((___S8*)___hp)+4+i32))=___U64UNBOX(val);
```

se trouve à la figure 3.23). On ajoute une branche dans le *switch* qui filtre les objets à traiter durant une collection pour séparer les enregistrements des autres types. Pour ce cas, on commence par marquer le descripteur de type. Celui-ci est toujours présent dans les enregistrements et contient l'information requise pour la collection du reste de l'objet. Il est important de commencer par celui-ci, car *mark_array* va mettre à jour les pointeurs dans le descripteur de type ce qui assure que les informations extraites sont correctes. En effet, même si, dans l'implantation actuelle du *garbage collector*, les valeurs des champs qui sont toujours vivants vont nécessairement être préservées durant une collection, leurs entêtes sont modifiés pour maintenir l'information de *forwarding* et la longueur des vecteurs y est stockée.

Une fois le traitement du descripteur de type fait, on appelle une fonction auxiliaire *scan_struct* (figure 3.24) qui reçoit en paramètre un pointeur vers le début des champs de l'enregistrement et un pointeur vers le corps du descripteur de type.

Cette fonction va appeler *mark_array* individuellement sur chacun des champs qui doivent être visités. Pour ce faire, on commence par extraire le vecteur de décalage approprié à la taille des mots du *runtime* par le biais de la macro *OFS_VECTOR*. Celle-ci ne fait qu'extraire le 8e ou 9e champ du descripteur de type. Une fois ce vecteur récupéré, on détermine sa longueur en l'extrayant de l'entête et en convertissant la valeur *boxer* vers un entier C.

On itère ensuite sur chacun des décalages en appelant *mark_array* au bon indice.

Figure 3.22 – *##type-type* étendu

```
(define ##type-type
 (let ((type
        '#(#f
           ##type-5
           type
           8
           #f
           #(id 1 #f name 5 #f flags 5 #f
             super 5 #f fields 5 #f
             scount 1 #f tcount 1 #f
             pos32 1 #f pos64 1 #f)
           9
           0
           #(4 8 12 16 20 24 28 32 36)
           #(8 16 24 32 40 48 56 64 72))))
   ; OK to mutate constant in Gambit
   (##structure-type-set! type type)
   (##subtype-set! type (macro-subtype-structure))
   type))
```

Figure 3.23 – Branche pour les enregistrements dans *scan*

```
\\ In scan(___pstate, *body)
\\ switch on subtype.
    case ___sSTRUCTURE:
      mark_array (___PSP body, 1);
      scan_struct(___PSP body+1, (___WORD *)___BODY(body[0]));
      break;
```

Seuls les champs ayant *SCMOBJ* comme type peuvent contenir des références donc on traite toutes les positions comme des pointeurs vers des mots. On ignore simplement les champs contenant des données numériques.

Figure 3.24 – Fonction *scan_struct*

```
#if ___WS == 4
#define OFS_VECTOR(type) (___WORD *)___BODY(___FIELD(type, 8))
#else
#define OFS_VECTOR(type) (___WORD *)___BODY(___FIELD(type, 9))
#endif

___HIDDEN void scan_struct
    (___PSD ___WORD* body, ___WORD* type)
{
  ___PSGET
  ___WORD* ofs = OFS_VECTOR(type);
  ___WORD ofs_length =
         (___WORD) ___INT(___VECTORLENGTH(ofs));

  int i;
  for (i = 0; i < ofs_length; ++i)
    {
      ___WORD index = ___INT(___FIELD(ofs, i));
      mark_array(___PSP ((___WORD*)
         (((___WORD) (body-1)) + index)), 1);
    }
}
```

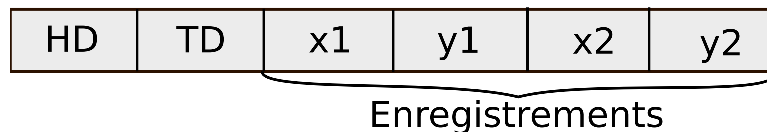
CHAPITRE 4

VECTEURS D'ENREGISTREMENTS

De nombreuses applications utilisant de grands ensembles de structures vont en allouer un nombre connu d'avance qui seront nécessaire tout au long du calcul. De plus, celles-ci seront fréquemment très homogènes (de quelques types distincts seulement). Une fois le calcul terminé, l'ensemble des enregistrements peut-être récupéré. On peut penser notamment à des simulations numériques ou des visualisations graphiques. Une représentation des enregistrements permettant l'allocation et la désallocation en masse ainsi qu'une représentation plus compacte de ces ensembles devient alors très avantageuse.

4.1 Survol

Figure 4.1 – Vecteur d'enregistrement



Afin de condenser la représentation des enregistrements, on implémente des vecteurs plats (qui contiennent directement les champs de leur contenu sans indirection) regroupant des enregistrements de même type (figure 4.1). Ceux-ci permettent dans un premier lieu de réduire la pression sur le gestionnaire de mémoire en allouant d'un seul coup un grand nombre d'enregistrements, ce qui évite de déclencher successivement des collections faisant grandir le tas par petits incréments. En les regroupant dans un seul objet, on permet aussi d'éliminer l'entête requis pour chacun des enregistrements en la remplaçant par l'entête du vecteur. Les descripteurs de type étant tous identiques par construction, il est aussi possible d'éviter de les ajouter à chacun des enregistrements en en plaçant une instance au début du vecteur. Les informations requises pour le parcours sont évidemment les mêmes que pour les enregistrements normaux et se trouvent donc dans le descripteur de type. Celui-ci est facilement accessible lors des collections.

Condenser de cette manière les enregistrements permet de réduire l'espace requis pour un ensemble d'enregistrements proportionnellement à leur nombre. En général, un ensemble de N enregistrements de taille M, occupera $N(M+2)$ mots machines si ceux-ci sont alloués individuellement alors qu'un vecteur d'enregistrements contenant N éléments de la même taille utiliserait seulement $NM+2$ mots. De plus comme l'allocation se fait par grand blocs, ces enregistrements seront allouer dans la région *___STILL* ce qui permet d'éviter d'avoir aussi à allouer assez d'espace pour les copier et donc d'économiser un autre facteur deux.

L'absence d'entête et de descripteur de type au niveau des enregistrements individuels et la présence de données quelconques à l'endroit où celles-ci auraient dû se trouver pose toutefois problème. Il devient nécessaire de pouvoir les distinguer autrement. Pour ce faire, on utilise le *tags* sur le pointeur (*tMEM2*) pour indiquer que la référence pointe vers un enregistrement contenu dans un vecteur d'enregistrements. Cette distinction permet d'éviter d'accéder à l'entête manquant ou d'essayer de parcourir individuellement les enregistrements contenus durant une collection. Récupérer un pointeur vers le descripteur de type reste problématique. On règle cette difficulté en introduisant un arbre de conteneurs utilisant l'adresse de l'enregistrement comme clef.

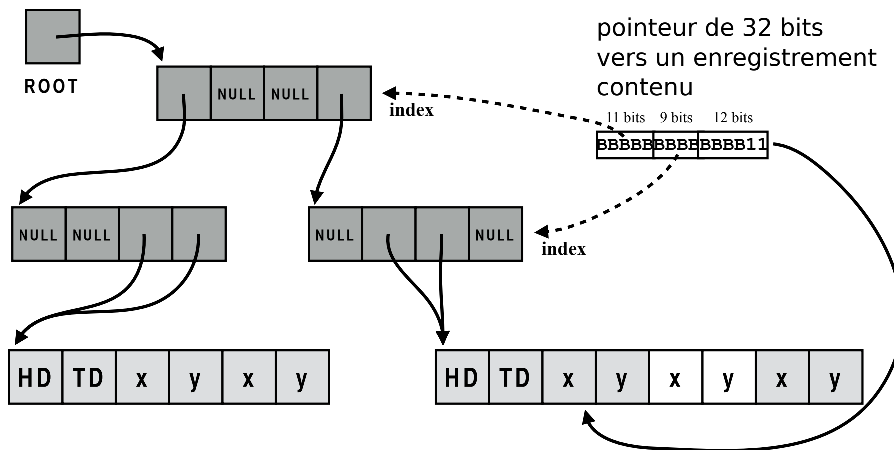
Pour permettre l'utilisation des vecteurs d'enregistrements, on modifie *define-type* pour générer des définitions spécialisées d'allocation, d'accès et de mutation dans les vecteurs d'enregistrements. On introduit aussi des primitives faisant l'implémentation des fonctions et structures requises et on modifie le *garbage collector* pour traiter les enregistrements contenus correctement.

4.2 Arbre de conteneurs

Les vecteurs d'enregistrements permettent d'éliminer les descripteurs de type redondant, mais comme certaines opérations requièrent d'y accéder, il est nécessaire de maintenir l'information quelque part. Pour ce faire, on a recours à un arbre de recherche couvrant toute la mémoire (figure 4.2). Cet arbre de recherche est adressé par la valeur du pointeur vers un objet contenu dans un vecteur d'enregistrements. On indexe chaque

niveau de l'arbre avec une partie des bits de ce pointeur afin de retrouver une référence vers le vecteur d'enregistrements. Le premier champ de celui-ci contient le descripteur de type des enregistrements contenus.

Figure 4.2 – Arbre de conteneurs



Toutes les références omises pointent vers *Null*.

Afin de maintenir des accès en temps constant, il est important que l'arbre ait un nombre fixe de niveaux. Le nombre choisi va évidemment varier selon la taille des mots machines pour éviter une trop grande granularité ou une explosion de la taille de l'arbre. À chaque fois que l'on va ajouter un vecteur d'enregistrements, on va parcourir toutes les feuilles de l'arbre correspondant à l'espace d'adressage sous-tendu par le vecteur pour y insérer un pointeur vers le vecteur. On n'a pas à se soucier de potentiel déplacement du vecteur, car ceux-ci sont toujours alloués de façon *still*.

4.2.1 Paramètres

Une série de constantes C paramètrent cet arbre (figure 4.3). Le découpage des adresses et leur niveau correspondant sont donnés dans la figure 4.2.

Afin de minimiser la taille de l'arbre en 32 bits, on divise les bits du pointeur en 3 sous-sections en plus des deux bits non utilisés. On utilise un bloc indexant les 11 premiers bits pour le premier niveau de l'arbre puisqu'il est profitable de réduire la taille des blocs du niveau suivant. En effet, le premier niveau est unique alors que le second niveau va

Figure 4.3 – Paramètres de l'arbre de conteneurs

```
#if ___WORD_WIDTH == 32
#define ___LOW_AD 9
#define ___HIGH_AD 11
#define ___LOW_GRAN ___LOW_AD
#else
#define ___LOW_AD 40
#define ___HIGH_AD 11
#define ___LOW_GRAN 10
#endif
```

comporter au moins deux blocs : au moins un pour les vecteurs d'enregistrements et un autre partagé qui indique que ce bloc est inutilisé. La nécessité d'allouer un bloc complet pour cet indicateur vient du désir d'avoir une opération d'accès en temps constant.

Les 9 bits suivants sont donc assignés au deuxième niveau d'indirection et on s'assure à l'allocation que les vecteurs d'enregistrements ont toujours une taille alignée sur un kilo-octet, ce qui permet de ne pas avoir à traiter les 12 derniers bits de l'espace d'adressage.

En 64 bits, la taille des blocs croissant de manière exponentielle avec le nombre de bits, on doit utiliser plus de niveaux. On conserve des tailles pour chaque bloc d'environ 1 à 2 kilo-octets et une taille alignée sur 2 kilo-octets pour les vecteurs ce qui nous donne 5 niveaux d'indirection (un de 11 bits et les autres de 10).

Figure 4.4 – Variables globales dans mem.c

```
___WORD *** ROOT = NULL;
___HIDDEN ___WORD ** null_block = NULL;
___HIDDEN int cont_count = 0;
```

L'arbre de conteneurs est référé par une variable *ROOT* (figure 4.4) accessible uniquement dans *mem.c*. Celle-ci est une valeur C initialisée au moment de la construction du premier vecteur d'enregistrement. Elle n'est pas exposée directement aux primitives Scheme afin d'éviter qu'elle ne soit tracée par le *garbage collector*. On veut aussi s'assurer qu'elle soit unique puisqu'une instance couvre l'ensemble de la mémoire.

Le *null_block*, quant à lui, est la variable référant au bloc ne contenant aucune référence vers un vecteur. Il est simplement représenté par la valeur *NULL*. On maintient aussi un compteur global (*cont_count*) des vecteurs d'enregistrements contenus dans l'arbre dans le but de pouvoir aisément récupérer l'espace pris par l'arbre de conteneurs si celui-ci n'est pas utilisé.

Figure 4.5 – Constantes liées à l'arbre de conteneurs

```
#define ___TYPE_TREE_LEVEL \
        ((___LOW_AD / ___LOW_GRAN) + \
        (___LOW_AD % ___LOW_GRAN))

#define ___LOW_SIZE (1 << ___LOW_GRAN)

#define ___HIGH_SIZE (1 << ___HIGH_AD)

#define ___HIGH_OFS (___WORD_WIDTH - ___HIGH_AD)

#define ___LOW_OFS (___HIGH_OFS - ___LOW_AD)

#define ___GRANULARITY (1 << (___LOW_OFS-2))
```

La constante *TYPE_TREE_LEVEL* (figure 4.5) représente le nombre de niveaux intermédiaires dans l'arbre de conteneurs. *LOW_SIZE*, *HIGH_SIZE* correspondent à la taille des blocs intermédiaires et du plus haut niveau respectivement. *LOW_OFS* et *HIGH_OFS* fournissent l'indice où on peut récupérer les bits dans le mot pour les deux types de blocs. La granularité (*GRANULARITY*) est la valeur sur laquelle la fin de chaque vecteur d'enregistrements doit être alignée. Elle dépend du nombre de bits utilisé à chacun des autres niveaux de l'arbre et du nombre de bits de *tagging*.

La macro *NEW_BLOCK* (figure 4.6) alloue un bloc ayant un nombre *size* de mots sans passer par la logique d'allocation de Scheme. *GET_ADDR* est utilisé pour obtenir

Figure 4.6 – Macros liées à l'arbre de conteneurs

```
#define ___NEW_BLOCK(size) \
    ((___WORD *) ___mem_alloc((size) * ___WS))

#define ___GET_ADDR(ptr) \
    ((___WORD) (___UNTAG((ptr)) + ___BODY_OFS))
```

l'adresse contenue dans une référence Scheme dans le but de trouver la bonne cellule dans l'arbre de conteneurs.

La recherche du vecteur d'enregistrement contenant une adresse se fait de façon distincte en 32 et 64 bits puisque les arbres n'ont pas la même profondeur et qu'on veut des opérations fixes. La macro *GET_CONTAINER* est donc définie en fonction de la taille des mots. Pour déterminer la taille d'un des niveaux inférieurs, on fournit comme interface une macro *GET_LOW_OFS*.

Figure 4.7 – Macros spécifiques en 32 bits

```
#define ___GET_LOW_OFS(n) ___LOW_OFS

#define ___GET_CONTAINER(add) \
        (ROOT[(add)>>___HIGH_OFS] \
        [((add)>>___LOW_OFS) & 0x1FF])
```

En 32 bits (figure 4.7), puisque l'arbre n'a que deux niveaux, la taille du niveau inférieur est simplement donnée par le nombre de bits servant de clefs.

La macro *GET_CONTAINER* quant à elle reçoit en argument l'adresse. Elle extrait les bits du haut en décalant à droite toute l'adresse. Ceci fait, on utilise la variable *ROOT* contenant la racine de l'arbre comme référence vers le bloc supérieur. On indexe à la valeur calculée dans le bloc pour obtenir un pointeur vers le bloc inférieur approprié.

Afin de masquer les 11 bits du haut pour extraire la portion des bits du bas, on utilise le masque 0x1FF, qui conserve seulement les 9 bits du bas, après les avoir décalés à droite de *LOW_OFS*. On utilise cette valeur comme indice dans le bloc pour obtenir une

référence vers une référence vers le vecteur d'enregistrement.

Figure 4.8 – Macros spécifiques en 64 bits

```
#define ___GET_LOW_OFS(n) \
        (___WORD_WIDTH - (n*___LOW_GRAN) - ___HIGH_AD)

#define ___GET_CONTAINER(add) \
        (((___WORD ******)ROOT)[(add)>>___HIGH_OFS] \
        [((add)>>(___GET_LOW_OFS(1))) & 0x3FF] \
        [((add)>>(___GET_LOW_OFS(2))) & 0x3FF] \
        [((add)>>(___GET_LOW_OFS(3))) & 0x3FF] \
        [((add)>>(___GET_LOW_OFS(4))) & 0x3FF])
```

En 64 bits (figure 4.8), comme il y a plus d'un niveau inférieur, on calcule le décalage requis pour isoler les bits de la partie appropriée de l'adresse au niveau *n* en fonction des autres paramètres.

De même, pour *GET_CONTAINER*, le calcul reste le même pour traverser le niveau supérieur, mais le masque doit être ajusté à la taille un peu plus grande des niveaux inférieurs (10 bits plutôt que 9) et on a 5 niveaux d'indirection au total.

Figure 4.9 – Macros parcourant l'arbre de conteneurs

```
#define ___CONTAINED_TYPE(add) \
         (((___WORD *)(___GET_CONTAINER((add))))[0])

#define ___GETCONTTYPE(ptr) \
         (___CONTAINED_TYPE(___GET_ADDR(ptr)))

#define ___CONTAINER_TYPE(con) \
         (((___WORD *)(___GET_ADDR(con)))[0])
```

Ces macros sont ensuite utilisées pour définir des macros extrayant les descripteurs de types de données contenues et des vecteurs. *CONTAINED_TYPE* (figure 4.9) retourne un pointeur vers le descripteur de type d'une adresse C contenue. *GETCONTTYPE* fait la même chose, mais en extrayant l'adresse avec *GET_ADDR*. Finalement, *CONTAINER_TYPE* retourne un pointeur vers le descripteur de type d'un vecteur d'enregistrement

dont l'adresse C est passée en argument.

4.2.2 Initialisation

Figure 4.10 – Fonction *init_type_tree*

```
___HIDDEN void init_type_tree ()
{
  int i;

  ROOT = malloc(___HIGH_SIZE * ___WS);

  null_block = NULL;

  for(i = ___HIGH_SIZE; i>=0; -i) ROOT[i] = null_block;
}
```

L'initialisation se fait au moment de la création d'un nouveau vecteur d'enregistrement si la racine *ROOT* est à la valeur par défaut (*NULL*). Elle se fait par le biais de la méthode *init_type_tree* (figure 4.10) qui allouer et initialiser un arbre vide où tous les champs pointent vers le bloc nul. Le bloc supérieur sera alloué avec un appel à malloc et tous ces champs seront ensuite initialisés à *NULL*.

4.2.3 Ajout

Lors de l'allocation des vecteurs d'enregistrements, il faut systématiquement ajouter les plages d'adresse correspondant au vecteur dans l'arbre de conteneurs.

L'ajout des vecteurs à l'arbre de conteneurs se fait à travers une fonction, *add_container* (figure 4.11), accessible uniquement dans *mem.c*. Celle-ci prend un pointeur vers le vecteur en argument. Elle extrait l'entête et la taille du vecteur pour déterminer les adresses de début et de fin à ajouter dans l'arbre.

Ces bornes sont ensuite utilisées pour calculer les indices entre lesquels on doit parcourir le niveau supérieur. On itère ensuite entre ces indices en ajoutant des blocs inférieurs au besoin et on utilise la macro *GET_CONTAINER* pour aller à toutes les cellules appropriées écrire l'adresse du vecteur.

Figure 4.11 – Fonction *add_container*

```
___HIDDEN void ___add_container (___WORD * obj)
{
  ___WORD head = obj[-1];
  ___WORD len = ___HD_WORDS(head);

  // Add blocks to fit the container.
  ___WORD begin = (___WORD) obj;
  ___WORD end = (___WORD) (obj+len);

  ___WORD add = begin >> ___HIGH_OFS;
  ___WORD limit = end >> ___HIGH_OFS;

#if ___WS == 8
  ___WORD ptr = add;
  while (ptr <= limit)
    {
      ___WORD *nb = (___WORD *) ROOT[ptr],
              *ns = (___WORD *) begin,
              *ne = (___WORD *) end;
      if (nb == null_block)
        {
          nb = ___NEW_BLOCK(___LOW_SIZE);
          int i;
          for(i = ___LOW_SIZE; i>=0; -i) nb[i] = null_block;
        }

      if (ptr != add) ns = NULL;
      if (ptr != limit) ne = NULL;

      add_type_block(nb, ns, ne, ___LOW_AD);
      ROOT[ptr] = (___WORD **) nb;
      ++ptr;
    }
#endif

#if ___WS == 4
  while(add <= limit) {
    if (ROOT[add] == null_block){
      ROOT[add] = new_block(___LOW_SIZE);
    }
    ++add;
  }
#endif

  // Find the slot that correspond to the address
  // of contained objects and set them to the type descriptor.
  while (end >= begin) {
    ___GET_CONTAINER(end) = (___WORD *) obj;
    -end;
  }
  ++cont_count;
}
```

L'ajout de blocs est très simple en 32 bits, car comme il n'y a qu'un niveau inférieur, on peut ajouter un nouveau bloc à chaque fois qu'une cellule visitée au niveau supérieur pointe vers *null_block*. Avec plusieurs niveaux toutefois, le chemin vers l'espace sous-tendant deux vecteurs peux diverger à n'importe quelle hauteur dans l'arbre.

En 64 bits, on doit faire appel à une fonction récursive pour gérer les branchements possibles. Celle-ci, *add_type_block*, va être appelée sur chacune des cellules du niveau supérieur entre l'adresse la plus petite et la plus grande du vecteur d'enregistrement. S'il s'agit d'une cellule limitrophe, on lui fournit l'adresse exacte de la borne inférieure ou supérieure selon le cas, sinon la valeur *NULL* comme indicateur. On teste aussi pour déterminer si le bloc référencé par le niveau supérieur est le bloc nul. Si ce n'est pas le cas, par exemple si un autre vecteur est alloué dans la même page, on réutilise le même bloc, sinon on en alloue un nouveau et on l'initialise à *null_block* pour garantir que chaque chemin à la même longueur.

La fonction *add_type_block* (figure 4.12) gère les branchements nécessaires durant l'ajout de vecteurs d'enregistrements à l'arbre de conteneurs. Elle prend en argument un pointeur vers un bloc, les pointeurs de débuts et de fins de la plage couverte dans ce niveau par le vecteur qu'on ajoute et un entier représentant le niveau dans l'arbre. Si la plage d'adresses que l'on doit traiter dans le bloc débute au début du bloc parce qu'il ne s'agit pas du premier bloc contigu, on passe plutôt la valeur *NULL* comme indicateur. L'argument *end* est utilisé de la même manière, mais avec le dernier bloc. L'argument *level* qui dénote le nombre de bits du mot restant à traiter (40 initialement et 10 de moins par appel récursif).

Le traitement se fait en quatre étapes. Premièrement, on teste si on a atteint les feuilles de l'arbre en testant si *level* est toujours plus grand que *LOW_GRAN*. S'il ne reste plus de bits à traiter, la récursion est terminée. Ensuite, on calcule les indices où commence et finit la plage d'adresses qu'il faut ajouter dans l'arbre. On calcule aussi la valeur appropriée du bloc nul en fonction du niveau.

Le calcul de ces indices se fait en testant premièrement si l'argument donnant le début ou la fin est *NULL*. Si c'est le cas, comme la plage d'adresses débutait avant ou après le bloc courant, on fixe la variable au début ou à la fin du bloc. Sinon on doit calculer

Figure 4.12 – Fonction *add_type_block*

```
___HIDDEN void add_type_block
___P((___WORD * block, ___WORD *start, ___WORD *end, int level),
     (block, start, end, level)
     ___WORD *block;
     ___WORD *start;
     ___WORD *end;
     int level;)
{
  if (level <= ___LOW_GRAN) return;
  ___WORD begin, limit, block_start = 0, block_end = ___LOW_SIZE -1;
  int i;

  if (start == NULL) begin = block_start;
  else begin = (((___SIZE_T) start) << (___HIGH_AD + ___LOW_AD - level))
              >> (___WORD_WIDTH - ___LOW_GRAN);

  if (end == NULL) limit = block_end;
  else limit = (((___SIZE_T) end) << (___HIGH_AD + ___LOW_AD - level))
              >> (___WORD_WIDTH - ___LOW_GRAN);

  ___WORD ptr = begin;
  ___WORD *nb = (___WORD *) block[ptr], *ns = start, *ne = end;
  if (is_null_block(nb))
    {
     nb = ___NEW_BLOCK(___LOW_SIZE);
     for(i = ___LOW_SIZE-1; i>=0; –i) nb[i] = (___WORD) null_block;
    }
  if (start != NULL && ptr != begin) ns = NULL;
  if (end != NULL && ptr != limit) ne = NULL;

  add_type_block(nb, ns, ne, level - ___LOW_GRAN);
  block[ptr++] = (___WORD) nb;

  if (ptr < limit) {
    nb = ___NEW_BLOCK(___LOW_SIZE), ns = start, ne = end;
    for(i = ___LOW_SIZE-1; i>=0; –i) nb[i] = (___WORD) null_block;
    if (start != NULL && ptr != begin) ns = NULL;
    if (end != NULL && ptr != limit) ne = NULL;
    while (ptr < limit)
      {
        if (is_null_block(nb))
           add_type_block(nb, ns, ne, level - ___LOW_GRAN);
        block[ptr++] = (___WORD) nb;
      }
  }
  if (begin != limit) {
    ptr = limit, nb = (___WORD *) block[ptr], ns = start, ne = end;
    if (is_null_block(nb))
      {
        nb = ___NEW_BLOCK(___LOW_SIZE);
        for(i = ___LOW_SIZE-1; i>=0; –i) nb[i] = (___WORD) null_block;
      }
    add_type_block(nb, ns, ne, level - ___LOW_GRAN);
    block[ptr] = (___WORD) nb;
  }
}
```

l'endroit où la plage commence ou se termine. On utilise le mot passé en paramètre. Celui-ci est en général l'adresse du vecteur ou cette adresse additionnée à la longueur du vecteur. On décale à gauche la valeur de la somme des bits considérés (tant haut que bas) moins le *level* pour éliminer tous les bits plus à gauche que ceux indexant le niveau courant. Pour extraire seulement les bits voulus, on décale ensuite du nombre de bits total moins le nombre de bits qu'on veut. On obtient ainsi seulement l'index approprié.

On fait ensuite une boucle entre les indices où on alloue si nécessaire un nouveau bloc initialisé au bloc nul. On détermine aussi si on doit propager encore les valeurs *start* et *end* ou les remplacer par *NULL* selon qu'ils sont le premier ou dernier élément de la boucle et leur valeur actuelle. On fait ensuite un appel récursif sur le bloc d'en dessous à l'indice courant en décrémentant le niveau et on inscrit l'adresse du bloc dans la cellule. On n'a pas besoin de valeur de retour, car la variable ROOT va contenir le niveau supérieur.

4.2.4 Retrait

Lorsque le *garbage collector* récupère un vecteur d'enregistrement inutilisé, il est important de retirer toutes les références vers le vecteur se trouvant dans les feuilles de l'arbre et de faire en sorte que les blocs qui ne sont plus sur un chemin menant vers un vecteur soient désalloués et remplacés par des références au bloc nul. Pour ce faire, on ajoute au code récupérant les objets de type *still* un appel conditionnel (si le *still* est un vecteur d'enregistrement) à la fonction *free_container* (figure 4.13).

Celle-ci commence, comme pour l'ajout de vecteurs, par extraire la longueur de celui-ci. On calcule ensuite les bornes de la façon usuelle. Ceci fait, on vérifie s'il s'agit du seul vecteur restant à l'aide de la variable *cont_count* qu'on décrémente afin de garder le compte à jour. S'il s'agit du dernier, on libère tout l'arbre de conteneurs en appelant *free_type_block* sur chaque cellule du niveau supérieur puis on le libère lui aussi. On libère aussi manuellement le bloc nul au cas où aucune référence ne resterait dans le niveau supérieur de l'arbre. On met ensuite *null_block* et *ROOT* à la valeur par défaut.

Dans le cas où il y a plus d'un vecteur d'enregistrement dans l'arbre, on doit faire attention à ne pas désallouer un bloc utilisé par un autre vecteur. On parcourt donc le

Figure 4.13 – Fonction *free_container*

```
___HIDDEN void ___free_container (void* ptr)
{
  ___WORD head = ((___WORD *) ptr)[-1];
  ___WORD len = ___HD_WORDS(head);
  ___WORD begin = ((___WORD *) ptr)+1;
  ___WORD end = ((___WORD *) ptr)+len;
  ___WORD add = begin >> ___HIGH_OFS;
  ___WORD limit = end >> ___HIGH_OFS;

  if (-cont_count == 0) {

    // Free everything
    ___WORD i = 0;

    while (i < ___HIGH_SIZE)
      {
        if (ROOT[i] != null_block)
          ___free_type_block(ROOT[i]);

        ++i;
      }

    free(ROOT);

    ROOT = NULL;

    ___free_null_block(null_block);

    null_block = NULL;

    return;
  }
// Iterate on blocks from begin to end and
// remove them if they were only used by ptr.

  while (limit >= add) {

    int tag = ___check_branching(ptr, ROOT[add]);

    if (tag) {
      ___free_type_block(block);

      ROOT[add] = null_block;
    }
    ++add;
  }
}
```

niveau supérieur entre les adresses assignées au vecteur que l'on souhaite retirer. Pour chacun on appelle une fonction auxiliaire qui vérifie si le vecteur est le seul présent dans le sous-arbre : *check_branching* (figure 4.14).

Figure 4.14 – Fonction *check_branching*

```
___HIDDEN int ___check_branching (void* ptr, ___WORD* block)
{
  ___WORD *i = block, *limit = block + ___LOW_SIZE -1;

  while (i != NULL || ((___WORD) i) & 0x1)
    {
      while (is_null_block(*i) || i > limit) ++i;
      i = *i;
    }

  if (i != ptr) return 0;

  i = limit;

  limit = block;

  while (i != NULL || ((___WORD) i) & 0x1)
    {
      while (is_null_block(*i) || i < limit) -i;

      i = *i;
    }

  return (i == ptr);
}
```

Cette fonction exploite le fait que les vecteurs d'enregistrements sont nécessairement formés de cellules contiguës et qu'il ne peut donc y avoir un autre vecteur à l'intérieur des bornes qu'ils définissent. Si un autre vecteur se trouve sous-tendu par le même bloc, il doit nécessairement être soit avant ou après celui que l'on enlève.

On cherche dans un niveau le premier champ non nul (il doit y en avoir au moins un) et on descend d'un niveau dans ce nouveau bloc. On répète tant qu'on n'a pas atteint les feuilles. Une fois ceci fait, on vérifie que le premier pointeur qu'on rencontre depuis la gauche est bien le vecteur retiré. S'ils diffèrent, on retourne faux, sinon, on recommence en cherchant de droite à gauche cette fois. On retourne ensuite vrai s'il n'y a aucun

branchement vers un autre vecteur d'enregistrement et faux sinon.

Figure 4.15 – Fonction *free_type_block*

```
___HIDDEN void ___free_type_block (void* ptr)
{
  // If it's tagged (scmobj) or NULL, recursion is done.
  if (((___WORD) ptr) & 0x1 || ptr == NULL) return;

  int  i = 0;

  while (i < ___LOW_GRAN)
    {
      if (!is_null_block(ptr[i]))
          ___free_type_block(((___WORD ***) ptr)[i]);

        ++i;
    }

  free(ptr);
}
```

Une fois qu'on a déterminé qu'un sous-arbre doit être désalloué, on utilise la fonction *free_type_block* (figure 4.15) pour désallouer le sous-arbre. Pour ce faire, on passe sur chaque cellule d'un bloc et si elle ne contient pas un pointeur vers le bloc nul, on appelle récursivement la fonction de désallocation sur le pointeur. On s'arrête lorsque l'on a atteint le niveau inférieur et on libère chaque bloc avant de retourner.

Une version spécialisée de *free_type_block*, *free_null_block*, est utilisée pour libérer le bloc nul puisque *free_type_block* évite de le faire. Celle-ci est un peu plus simple, car comme toutes les références dans un niveau sont identiques, on n'a pas à itérer sur chaque cellule.

4.3 Enregistrements contenus

Les vecteurs d'enregistrements contiennent, évidemment, des enregistrements. On veut toutefois éliminer les descripteurs de types individuels pour les consolider dans un descripteur de type au début du vecteur. On veut aussi se débarrasser complètement de l'entête des mots puisque l'information requise se trouve dans le descripteur de type et

l'entête du vecteur. Il est donc nécessaire de prendre en compte le besoin de le récupérer dans l'arbre de conteneurs lorsque c'est nécessaire. Il est aussi souhaitable que ces enregistrements soient utilisables comme n'importe quels autres enregistrements de façon totalement transparente. On doit donc modifier les tests de types pour faire des recherches dans l'arbre si le type d'un objet est celui d'un objet contenu.

Pour ce faire, on commence par redéfinir l'étiquette *tMEM2* (les références de types 2). Celle-ci était précédemment utilisée pour distinguer les paires sans devoir récupérer l'entête. On l'utilise plutôt pour distinguer les références internes dans un vecteur d'enregistrement.

Figure 4.16 – Primitive *##structure-type*

```
(define-prim (##structure-type obj)
  (if (##contained? obj)
      (##contained-type obj)
      (##vector-ref obj 0)))
```

On modifie ensuite la fonction primitive *##structure-type* (figure 4.16) qui retourne le descripteur de type d'un enregistrement. On commence par vérifier si l'objet passé en argument est contenu avec la fonction *##contained ?* Si c'est le cas, on va utiliser la primitive *##contained-type* qui se traduit par un appel à la fonction C *contained_type*. Celle-ci fait directement un appel à la macro *GETCONTTYPE*, mais il est nécessaire de passer par une fonction C définie dans *mem.c* afin que *ROOT* soit visible. Dans le cas d'un enregistrement normal, on peut récupérer son descripteur de type en accédant au premier champ.

Le test *##contained ?* (figure 4.17), quant à lui, se traduit directement vers la macro C *ISCONTAINED* qui vérifie que l'objet passé en paramètre a le marqueur de type *tMEM2*. Comme les bits indiquant le type sont à même le pointeur, pas besoin de traitements autres.

Figure 4.17 – Code généré pour *##contained ?*

```
#define ___ISCONTAINED(o) ___TESTTYPE(o,___tMEM2)
```

Figure 4.18 – Primitive *##structure-direct-instance-of ?*

```
(define-prim (##structure-direct-instance-of? obj type-id)
  (and (or (##contained? obj) (##structure? obj))
       (##eq? (##type-id (##structure-type obj))
              type-id)))
```

On fait des modifications semblables aux tests de types *##structure-instance-of ?* et *##structure-direct-instance-of ?* (figure 4.18) en ajoutant un test pour savoir si le pointeur est contenu au lieu d'une vérification du sous-type dans l'entête pour déterminer si l'objet est bien un enregistrement.

Figure 4.19 – Code généré pour *##structure-type* et *##structure-direct-instance-of ?*

```
#define ___STRUCTURETYPE(x) \
        (___ISCONTAINED(x)? \
                ___contained_type(___PSTATE, x):\
                ___VECTORREF(x,___FIX(0)))

#define ___STRUCTUREDIOP(x,typeid)\
        ((___ISCONTAINED(x)|| \
        ___TESTSUBTYPE(x,___sSTRUCTURE))&& \
        ___TYPEID(___STRUCTURETYPE(x))==(typeid))
```

Des modifications analogues sont aussi, bien entendu, requises aux versions étendues *inline* de ces primitives. Dans ce cas, on utilise directement les macros servant à implémenter les tests déterminant si un objet est contenu. Les versions en macros C des primitives *##structure-type* et *##structure-direct-instance-of ?*, *STRUCTURETYPE* et *STRUCTUREDIOP* (figure 4.19) sont essentiellement une traduction directe vers C.

Figure 4.20 – Primitive *##structure-vector ?*

```
(define-prim (##structure-vector? o)
  (##c-code "___RESULT =
    ___ISCONTAINER(___ARG1)?___TRU:___FAL;" o))
```

Une primitive est aussi fournie pour déterminer si un objet est un vecteur d'enregistrement. Celle-ci, *##structure-vector ?* (figure 4.21), est implémentée par un appel à la macro *ISCONTAINER* à travers l'interface et est étendue *inline* vers ce même appel.

Figure 4.21 – Code généré pour *##structure-vector ?*

```
#define ___ISCONTAINER(o) ___TESTSUBTYPE(o,___sCONTAINER)
```

La macro en question ne fait qu'un banal test de sous-type vers le sous-type *sCONTAINER* (figure 4.22). Ce sous-type est ajouté spécifiquement pour supporter les vecteurs d'enregistrements, notamment durant la gestion de la mémoire, où il est crucial de pouvoir les distinguer précisément des autres objets. Comme l'ensemble des valeurs de sous-types possibles n'est pas complètement utilisé, on lui a simplement assigné la valeur 16.

4.4 Allocations

L'allocation des vecteurs d'enregistrements se fait par le biais d'une primitive ajoutée au *runtime*. Celle-ci, *alloc_container*, est appelée par la fonction primitive Scheme *##structure-vector* dont les appels sont construits à travers *define-type*, qui, lui, définit une fonction *make-<type>-vector* pour chaque type défini de manière analogue aux appels à *##structure*. Ainsi, lors de la définition d'un point, on créera l'appel montré dans la figure (figure 4.23).

Dans *##define-type-expand*, on construit l'appel par le biais de la fonction auxiliaire *generate-vector* (figure 4.23). Celle-ci construit un symbole correspondant au nom de la fonction qui sera générée, puis construit un appel à la primitive en lui fournissant le

Figure 4.22 – Exemple d'expansion de *make-point-vector*

```
(define point x)
    =>
(define (make-point-vector n)
    (##structure-vector <Type-point> 1 n))
```

descripteur de type et le nombre de mots contenu dans les champs de l'enregistrement (on soustrait 1 à *max-words*, car le descripteur de type ne sera pas répété dans chaque champ).

La primitive *##structure-vector* (figure 4.24) prend ensuite ces arguments et fait un appel à la fonction C *alloc_container* à travers l'interface *##c-code*. Si l'allocation ne retourne pas une erreur, on décrémente le compteur de référence sur l'objet *still* pour s'assurer qu'il sera bien pris en charge par le *garbage collector*. La fonction C prend en argument le nombre de mots total et le descripteur de type. On fait donc le calcul avant de faire l'appel.

Comme l'appel à la fonction d'allocation peut déclencher le *garbage collector*, on doit faire attention pour que les registres soient sauvegardés avant l'appel. Ceux-ci doivent être à jour, car ils seront traités comme des racines durant la collection.

La fonction d'allocation (figure 4.25) calcule le nombre de mots nécessaires pour avoir le bon alignement pour respecter les frontières entre les blocs dans l'arbre de conteneurs. Une fois le calcul fait, on trouve le nombre d'octets requis en fonction de la taille des mots et on fait une allocation avec *alloc_scmobj_still* qui retourne un pointeur vers l'objet *still* de type *sCONTAINER* nouvellement alloué.

On récupère ensuite l'adresse où débute le vecteur et on initialise tous les champs à 0 pour s'assurer qu'ils ne contiennent pas une adresse de pointeur Scheme valide qui pourrait être tracé par le *garbage collector*. On doit ensuite l'ajouter dans l'arbre de conteneurs. Pour ce faire, on s'assure que celui-ci existe. On l'initialise au besoin avec *init_type_tree* puis on utilise *add_container* pour ajouter la référence à l'arbre. On écrit aussi le descripteur de type au début du vecteur et on laisse le vecteur nouvellement créé aux soins de la gestion de mémoire habituelle avec *release_scmobj*.

Figure 4.23 – Construction de *make-<type>-vector*

```
(define (generate-vector)

        (let ((vector-name
                (##string->symbol
                  (##string-append "make-"
                                   (##symbol->string name)
                                   "-vector"))))

          (if macros?
              `((##define-macro (,vector-name n)
                  (##list '(let ()
                             (##declare (extended-bindings))
                             ##structure-vector)
                          ',type-expression
                          ,(##fixnum.- max-words 1)
                          n)))

              `((define (,vector-name n)
                  ((let ()
                     (##declare (extended-bindings))
                     ##structure-vector)
                   ,type-expression
                   ,(##fixnum.- max-words 1)
                   n))))))
```

4.5 Références et mutations

De la même manière, on ajoute des fonctions d'accès et de mutations pour les vecteurs d'enregistrements. On modifie *##define-type-expand* pour construire des fonctions spécialisés d'allocations qui vont appeller une primitive définie dans *_kernel.scm* et ajoutée à la liste des primitives dans *_prims.scm*. Celles-ci vont être traduis vers un ou des appels de macros C contenues dans *gambit.h.in*. Comme les mutations des cellules d'un vecteur d'enregistrement modifient un enregistrement complet, le code généré pour son extension est plus semblable à celui d'un allocateur d'enregistrements que d'un simple mutateur.

Les définitions de fonctions et macros construites par *define-type* sont générées par un appel aux fonctions auxiliaires *generate-vector-ref* (figure 4.26) et *generate-vector-set!*. Les deux font essentiellement la même chose que *generate-vector*. Ils construisent le nom de la fonction ou de la macro et l'utilisent pour construire la définition avec un appel à la

Figure 4.24 – Primitive *##structure-vector*

```
(define-prim (##structure-vector type size n)
  (##declare (not interrupts-enabled))
  (##c-code #<<end-of-code

    ___FRAME_STORE_RA(___R0)

    ___W_ALL

    ___SCMOBJ result =
        ___alloc_container(___ps, ___ARG1, ___ARG2);

    ___R_ALL

    ___SET_R0(___FRAME_FETCH_RA)

    if (!___FIXNUMP(result))
      ___still_obj_refcount_dec (result);

    ___RESULT = result;

end-of-code

    type
    (##fixnum.* n size)))
```

primitive approprié. L'appel à *##structure-vector-ref* va recevoir en argument le vecteur, la taille des enregistrements contenus et l'indice de la cellule voulue.

Dans le cas de *generate-vector-set !*, l'appel à *##structure-vector-set !* (figure 4.28) va requérir un peu plus d'informations. En effet, de la même manière qu'à l'allocation d'enregistrements typés, il est requis de fournir la liste des décalages tant en 32 qu'en 64 bits et une liste des attributs de chacun des champs pour générer la version *inline* du code.

Les références dans des vecteurs d'enregistrements sont essentiellement implémentées de la même manière que celles dans des vecteurs normaux. Elles diffèrent principalement par le fait que le décalage qu'on leur fournit doit prendre en compte la taille des enregistrements contenus. On fait simplement le calcul de l'indice (*i*) fois cette taille (*s*) dans la macro *CONTAINERREF* (figure 4.27). Les autres particularités sont que la référence retourne un pointeur vers la cellule appropriée plutôt que la valeur contenue et que ce pointeur va avoir le sous-type des objets contenus : *tMEM2*.

Figure 4.25 – Fonction *alloc_container*

```
___EXP_FUNC(___SCMOBJ,___alloc_container)
    (___processor_state ___ps,
     ___SCMOBJ type , ___SIZE_T words)
  {
    ___WORD n = ___INT(words) + 1;

    // Ensure the container is alligned
    // on a multiple of the granularity,
    ___WORD pad = (n % ___GRANULARITY);

    if (pad != 0)
      n += ___GRANULARITY - pad;

    ___SIZE_T bytes = n*___WS;

    ___SCMOBJ ptr =
        alloc_scmobj_still(___ps, ___sCONTAINER, bytes);

    ___WORD * this =  ___UNTAG((___WORD)ptr) + ___BODY_OFS;

    int i;

    for(i = 1; i<n; i++) this[i] = 0;

    if (ROOT == NULL)
        init_type_tree();

    this[0] = (___WORD) type;

    ___add_container(this);

    ___release_scmobj(ptr);

    return ptr;
  }
```

Figure 4.26 – Construction des définitions pour *<type>-vector-ref*

```
(define (generate-vector-ref)
        (let ((vector-name (##string->symbol
         (##string-append (##symbol->string name)
                          "-vector-ref"))))
          (if macros?
              `((##define-macro (,vector-name v i)
                  (##list '(let ()
                             (##declare (extended-bindings))
                             ##structure-vector-ref)
                          v
                          ,(##fixnum.- max-words 1)
                          i)))
              `((define (,vector-name v i)
                  ((let ()
                     (##declare (extended-bindings))
                     ##structure-vector-ref)
                   v
                   ,(##fixnum.- max-words 1)
                   i))))))
```

Figure 4.27 – Code généré pour *##structure-vector-ref*

```
#define ___CONTAINERREF(c,s,i) \
  (___SCMOBJ) ___TAG(((___WORD*) \
  ___UNTAG_AS(c,___tSUBTYPED))+
    (___INT(i) * ___INT(s)), ___tMEM2)
```

L'implantation des mutations d'enregistrements dans des vecteurs d'enregistrements est nécessairement un peu plus complexe. La primitive sans extension emphinline va, un peu comme pour l'allocation des enregistrements typés, utiliser la fonction *##struct-set!-by-code* en conjonction avec les attributs de chaque champ et leur décalage pour mettre à jour leurs valeurs. On utilise un *##structure-vector-ref* pour récupéré le pointeur vers la référence interne à "initialiser". Ce pointeur est à un décalage suffisant pour sauter par-dessus la cellule qui aurait dû contenir le descripteur de type dans le cas d'un enregistrement alloué normalement. Ceci permet de réutiliser les décalages tels quels.

L'extension *inline* de *##structure-vector-ref* se réduit simplement à la macro *CONTAINERREF*, mais la génération de *##structure-vector-set!* (figure 4.29) est plus complexe.

Figure 4.28 – Primitive *##structure-vector-set !*

```
(define-prim (##structure-vector-set!
        struc max-words i ftypes pos32 pos64 . fields)

  (let ((s (##structure-vector-ref struc max-words i)))

    (let loop ((f fields) (ft ftypes)
               (p32 pos32) (p64 pos64))

      (if (##pair? f)
          (let ((setter
            (##struct-set!-by-code (##car ft))))

            (setter s (##car f) (##car p32)
                  (##car p64) #f #f)
            (loop (##cdr f) (##cdr ft)
                  (##cdr p32) (##cdr p64)))))))
```

Une série d'appels à des mutateurs spécialisés pour des cellules de types hétérogènes requiert d'itérer sur chaque champ et de déterminer la macro à utiliser en fonction de l'information constante fournie par *define-type*. Un ensemble de macros *SET_<type>_TSTRUCTURE_ELEM* est donc défini à cet effet.

De la même façon que lors de la génération des allocations, on extrait les valeurs constantes puis on les utilise pour sélectionner la macro de mutation. Il n'y a évidemment pas de code à généré pour réserver la mémoire sur le tas ou retourner l'adresse de l'enregistrement puisque celle-ci est déjà allouée dans un vecteur. On itère sur chacun des champs pour construire les appels aux mutateurs puis on retourne simplement la constante *VOID*.

Ces macros de mutations (figure 4.30) prennent l'adresse du corps du vecteur d'enregistrements et calcule le décalage requis pour accéder au champ en sautant un nombre *i* d'enregistrements de taille *mw* et assigne la valeur *val* dans cette cellule. Notons que *mw* est une constante numérique passée directement dans le code C généré de même que *i32* et *i64*. Ils n'ont donc pas à être *unboxed*.

Figure 4.29 – Génération de l'expansion *inline* de *##structure-vector-set !*

```
(define (targ-apply-structure-vector-set)
  (targ-setup-inlinable-proc
   #f
   #f
   #f
   (lambda (opnds sn)
     (let* ((n     (length opnds))
            (st    (car opnds))
            (mw    (obj-val (list-ref opnds 1)))
            (i     (list-ref opnds 2))
            (tags  (obj-val (list-ref opnds 3)))
            (pos32 (obj-val (list-ref opnds 4)))
            (pos64 (obj-val (list-ref opnds 5)))
            (rest  (cdddr (cdddr opnds))))
       (let ()
         (define (add-element tag c i mw i32 i64 elem)
           (case (##fixnum.arithmetic-shift-right tag 4)
             ((1)  (list "SET_S8_TSTRUCTURE_ELEM"
                (targ-opnd c) (targ-opnd i) mw i32 i64
                        (targ-opnd elem)))
                ...
             ((5)  (list "SET_U8_TSTRUCTURE_ELEM"
                (targ-opnd c) (targ-opnd i) mw i32 i64
                        (targ-opnd elem)))
                ...
             ((9)  (list "SET_F32_TSTRUCTURE_ELEM"
                (targ-opnd c) (targ-opnd i) mw i32 i64
                        (targ-opnd-flo elem)))
             ((10) (list "SET_F64_TSTRUCTURE_ELEM"
                (targ-opnd c) (targ-opnd i) mw i32 i64
                        (targ-opnd-flo elem)))
             (else (list "SET_TSTRUCTURE_ELEM"
                (targ-opnd c) (targ-opnd i) mw i32 i64
                        (targ-opnd elem)))))

         (let loop ((tag tags) (p32 pos32) (p64 pos64) (r rest))
           (if (pair? r)
               (begin
                 (targ-emit
                        (add-element (car tag) st i mw
                                        (car p32) (car p64) (car r)))
                 (loop (cdr tag) (cdr p32) (cdr p64) (cdr r)))))
         (list "VOID")
         )))))
```

Figure 4.30 – Mutateur d'un élément *float32* dans un *##structure-vector-set!*

```
#define ___SET_F32_TSTRUCTURE_ELEM(c,i,mw,i32,i64,val) \
 *((___F32*)   (((___S8*)((___BODY(c)+ \
 (___INT(i)*(mw)))))+i64))=(val);
```

4.6 Modifications au *garbage collector*

L'ajout de vecteurs d'enregistrements possiblement typés aux définitions de types demande des modifications au *garbage collector* de nature similaire à l'ajout des types en plus de changements pour mettre à jour l'arbre de conteneurs lorsqu'on récupère un vecteur. Il est aussi crucial de garantir la survie de tout un vecteur tant qu'un des enregistrements qu'il contient reste en vie ce qui demande le traitement des données contenues lors du marquage.

Pour traiter le cas des références vers des enregistrements contenus, on se contente de remplacer lors du parcours d'un objet dans *mark_array* chaque référence vers objet de type *tMEM2* par une référence vers son conteneur à l'aide de la macro *GET_CONTAINER*. On s'assure ainsi que le conteneur va être marqué dès qu'un des enregistrements contenus est traversé et on évite d'avoir à dupliquer le code servant à leur gestion durant la collection. En effet, le vecteur ainsi marqué va être *scanné* d'un bloc au moment où l'on parcourt les objets *still* vivants.

C'est durant le parcours des objets *still* qui n'ont pas été marqués et qui seront récupérés qu'il faut tester si ceux-ci sont de sous-type *sCONTAINER* pour s'assurer d'appeler *free_container* systématiquement sur tous les vecteurs d'enregistrements réclamés afin de garantir la cohérence de l'arbre de conteneurs.

Afin de gérer des vecteurs d'enregistrements pouvant contenir des champs qui ne sont pas des objets Scheme, on doit ajouter un autre branchement à la fonction *scan* (figure 4.31). Dans ce branchement, on commence encore une fois par marquer le descripteur de type puisque l'on aura besoin de celui-ci pour parcourir les champs et on doit donc s'assurer que les pointeurs qu'il contient ont bien été mis à jour. On fait ensuite appel

Figure 4.31 – Branche des vecteurs d'enregistrements dans *scan*

```
// In a switch on subtypes.
case ___sCONTAINER:
  /* Scan type descriptor */
  mark_array(___PSP body, 1);
  scan_container(___PSP body+1,
    (___WORD *) ___BODY(body[0]) ,words-1);
  break;
```

à la fonction *scan_container*, à laquelle on passe un pointeur vers le premier champ du vecteur et un pointeur vers le descripteur de type et le nombre de mots que contient le reste du vecteur.

La fonction *scan_container* (figure 4.32) va, comme son nom l'indique, parcourir le vecteur d'enregistrement en marquant à l'aide de *mark_array* chacun des champs qui contiennent un objet Scheme et en sautant ceux qui contiennent des données C.

Pour ce faire, on récupère dans le descripteur de type le vecteur approprié de décalage à l'intérieur d'un des enregistrements contenus ainsi que sa longueur avec l'aide des macros *OFS_VECTOR* et *VECTORLENGTH* respectivement. On extrait aussi le nombre de mots assignés à un des enregistrements du descripteur de type et on utilise cette valeur et le nombre de mots total passé en paramètre pour obtenir le nombre d'enregistrements contenus.

Ensuite, pour chacun de ces enregistrements, on va boucler sur le vecteur des décalages et appeler *mark_array* sur les cellules susceptibles de contenir un pointeur valide. On met à jour le pointeur en l'avançant de la taille d'un des enregistrements après chacune de ces boucles.

Figure 4.32 – Fonction *scan_container*

```
___HIDDEN void scan_container
    (___PSD ___WORD* body, ___WORD* type, ___SIZE_TS len)
{
  ___PSGET

  ___WORD * ofs = OFS_VECTOR(type), *ptr = body;

  ___WORD ofs_length =
        (___WORD) ___INT(___VECTORLENGTH(ofs));

  ___WORD max_words = ___INT(___FIELD(type, 6));

  ___WORD scount = len / max_words;

  int i,j;

  for (i = 0; i<scount; ++i)
    {
      for (j = 0; j<ofs_length; ++j)
        {
          ___WORD index = ___INT(___FIELD(ofs, j));

          mark_array(___PSP ((___WORD*)
                (((___WORD) (ptr-1)) + index)), 1);
        }
      ptr += max_words;
    }
}
```

CHAPITRE 5

ÉVALUATION

5.1 Méthodologie

Afin d'évaluer d'importants aspects de l'impact sur la performance associé à l'implémentation d'enregistrements typés et de vecteurs d'enregistrements, nous avons utilisé une série de petits tests employant les opérations primitives associées aux enregistrements qui nous intéressent. Pour pouvoir comparer les performances, on utilise la nouvelle primitive en la comparant à une version faisant le même travail en utilisant les anciennes primitives.

On utilise le même compilateur dans les deux cas et on exécute un script roulant une série de programmes de tests sur la même machine (Intel Core i7 2.2 GHz, 8 GB DDR3). Les programmes de tests sont compilés avec la commande *./gsc/gsc - :=. -exe benchmarks/<fichier>.scm* à partir du répertoire principal de Gambit. Cette commande indique simplement que le répertoire à partir duquel les références des fichiers sont indiquées est le répertoire courant et qu'on souhaite générer un fichier exécutable binaire.

5.2 Déclarations

Pour ce faire, on compile chaque programme de test avec le compilateur souhaité. Chacun comporte un ensemble de déclarations (figure 5.1) visant à normaliser les performances pour utiliser uniquement la primitive testée dans chaque cas de façon efficace et identique. Ces programmes font évidemment toujours le même nombre d'itérations que leurs paires et s'exécutent en à peu près une seconde. On fait aussi attention de propager les variables retournées par les appels à chaque tour de boucle pour éviter que le compilateur n'élimine des appels en optimisant le code.

Les déclarations *standard-bindings* et *extended-bindings* permettent au compilateur de tenir pour acquis que les fonctions primitives standards et non standards respectivement ne sont jamais redéfinies, ce qui permet des substitutions directes par leurs définitions.

Figure 5.1 – Déclarations au début de chaque test.

```
(declare (standard-bindings) (extended-bindings)
         (fixnum) (not safe) (block) (inlining-limit 0))
```

L'option *fixnum* dicte au compilateur d'utiliser des entiers à précision fixe plutôt qu'une tour numérique à précision infinie. La déclaration *not safe* indique que le code qui suit peut utiliser les optimisations marquées non sécuritaires comme, par exemple, les spécialisations de fonctions vers des versions ne faisant pas de tests de typage. La déclaration *block*, quant à elle, indique que le programme est complètement contenu dans ce fichier, ce qui permet plusieurs optimisations, notamment aux boucles qui seront utilisées largement par les programmes de tests. L'*inlining-limit* fixe le facteur de croissance maximal du code à la suite des optimisations. On le fixe ici à 0 pour s'assurer que les boucles ne sont jamais déroulées pour garantir que la version compilée des deux programmes est équivalente.

On utilise aussi des définitions de types générant des macros (figure 5.2) pour éviter que les opérations sur des enregistrements n'aient un appel de fonction qui n'est pas optimisé autour du code *inline* alors que, par exemple, les primitives de vecteurs sont utilisées directement et donc étendues *inline* sans cet appel.

Le code généré va ressembler à celui retrouvé dans la figure 5.3. On a ici un appel à une fonction construisant un enregistrement contenant deux champs entiers signés de 32 bits assignés comme corps de la fonction *run*. Les opérations sont effectuées une seule fois par branchement et elles sont toutes composées de façon *inline*.

Ensuite, on utilise un script qui exécute en boucle le programme compilé sur 20 itérations et prend la moyenne géométrique des temps d'exécutions en enlevant le plus grand et le plus petit. On compare le temps moyen de deux versions et on prend le ratio de ces temps moyens.

Figure 5.2 – Code de *##structure-typed*

```
(declare (standard-bindings) (extended-bindings) (fixnum)
        (not safe) (block) (inlining-limit 0))

(define-type point id: point macros:
        (x int32:) (y int32:))

(define stop 10000000)

(define v (make-vector stop #f))

(define (run)

  (let loop ((i (- stop 1)) (result #f))

    (if (>= i 0)

      (loop (- i 1)
        (begin (##vector-set! v i (make-point 11 22)) v))

      result)))

(define s (##exec-stats run))
```

Figure 5.3 – Coeur de la boucle de *##structure-typed*

```
___DEF_GLBL(___L4_run)

   ___BEGIN_ALLOC_TSTRUCTURE(3)

   ___ADD_TSTRUCTURE_ELEM(0,0,___SUB(0))

   ___ADD_S32_TSTRUCTURE_ELEM(4,8,___FIX(11L))

   ___ADD_S32_TSTRUCTURE_ELEM(8,12,___FIX(22L))

   ___END_ALLOC_TSTRUCTURE(3)

   ___SET_R2(___GET_TSTRUCTURE(3))

   ___VECTORSET(___GLO_v,___R1,___R2)

   ___SET_R2(___GLO_v)

   ___SET_R1(___FIXSUB(___R1,___FIX(1L)))

   ___CHECK_HEAP(2,4096)
```

5.3 Enregistrements typés

5.3.1 Programmes de tests

Pour évaluer les performances des enregistrements typés vis-à-vis des enregistrements normaux, on s'intéresse aux temps d'allocation, d'accès et de mutation.

Par exemple, le programme *##structure* (figure 5.4) fait l'allocation d'une série de 10 millions de points en 2 dimensions contenant les entiers 11 et 22 dont les références sont maintenues dans un vecteur global. La primitive *##exec-stats* retourne simplement des statistiques sur l'exécution comme le temps, le nombre de déclenchements du *gc*, etc. Celles-ci sont utilisées par le script pour générer les temps d'exécution.

Les deux microprogrammes (figure 5.2 et 5.4) font dans ce cas exactement la même chose. La seule distinction est le format des données allouées et les opérations de stockage nécessaires pour initialiser les champs. Ceux-ci sont modifiés en ajoutant des annotations de types aux champs dans *define-type* (figure 5.5)

On fait une boucle très serrée décrémentant un compteur et propageant le résultat qui initialise l'indice courant du vecteur global à une référence vers l'enregistrement nouvellement alloué. On utilise un processus identique avec des enregistrements ayant respectivement 20 champs non typés et 20 champs *int8* dans le programme *##structure-20*. Le choix du nombre de champs est fait en suivant l'observation que, dans la base de code de Gambit, le nombre de champs d'un enregistrement complexe à plusieurs niveaux d'héritage, comme les ports de lecture et d'écriture, ne semble jamais dépasser vingt. Des enregistrements ayant un nombre de composantes distinctes plus grand semblent très rares.

Pour tester l'influence de la taille du tas sur les temps d'allocations, on refait le test précédent en utilisant un tas initial de deux giga-octets dans le programme *##structure-20-big-heap*. L'augmentation de la taille du tas initial permet de réduire le nombre de déclenchements du *garbage collector* à un seul durant une exécution pour chacun des deux programmes, comparativement à 4 et 6 respectivement pour les versions typées et non typées avec le tas par défaut. L'entrée *##structure-20-big-gc* compare les temps passés en collection dans le programme *##structure-20-big-heap*.

Figure 5.4 – Code de *##structure*

```
(declare (standard-bindings) (extended-bindings) (fixnum)
        (not safe) (block) (inlining-limit 0))

(define-type point id: point macros: x y)

(define stop 10000000)

(define v (make-vector stop #f))

(define (run)

  (let loop ((i (- stop 1)) (result #f))

    (if (>= i 0)

      (loop (- i 1)
        (begin (vector-set! v i (make-point 11 22)) v))

      result)))

(define s (##exec-stats run))
```

Figure 5.5 – Code de *##structure-typed*

```
(define-type point id: point macros:
        (x int32:) (y int32:))
```

On procède de manière identique pour les tests d'accès et de mutation. On compare les temps requis pour récupérer une valeur dans une cellule soit d'un vecteur soit d'un enregistrement typé ou pour la modifier. Les méthodes utilisées sont dans les deux cas celles définies par *point-x* et *point-x-set !*.

On utilise aussi deux programmes un peu plus longs : *distance sort* (figure 5.6) et *convex envelope*. Le premier fait le tri sur place d'un vecteur de 20 000 points en deux dimensions par rapport à la distance euclidienne en utilisant un tri par sélection. Ce programme dénote peu de gains d'espace en compactant les champs et fait beaucoup d'accès aux champs.

L'autre calcule l'enveloppe convexe d'un nuage aléatoire de 4 millions de points bidimensionnels en utilisant l'algorithme de Jarvis [**?**].

5.3.2 Résultats

Les résultats de ces exécutions sont présentés dans le tableau 5.I.

Tableau 5.I – Résultats avec des enregistrements typés

Typed structures primitives			
name	baseline (s)	typed (s)	(ratio)
##structure	2.88	2.56	(**.89**)
##structure-20	11.12	2.13	(**.19**)
##structure-20-gc	10.64	1.76	(**.17**)
##structure-20-big-heap	7.10	2.13	(**.30**)
##structure-20-big-gc	4.98	1.74	(**.35**)
##unchecked-structure-ref	2.15	2.21	(**1.03**)
##unchecked-structure-set	2.43	2.56	(**1.05**)
##unchecked-uint8-structure-ref	2.30	2.43	(**1.06**)

La colonne *ratio* présente le ratio $typed/vector$. Ceux-ci sont respectivement la moyenne géométrique des temps en secondes pris par les programmes de tests.

On constate rapidement que, dans les tests d'allocation, les programmes utilisant des enregistrements typés sont tous plus rapides que leurs homologues utilisant des enregistrements normaux. Toutefois l'inverse est vrai pour tous les programmes testant l'accès aux champs d'un enregistrement. Les variations des ratios vont de 17% à 114%,

Figure 5.6 – Code de *distance-sort int32*

```
(declare (standard-bindings) (extended-bindings) (fixnum)
    (not safe) (block) (inlining-limit 0))
(define-type point id: point macros: (x int32:) (y int32:))
(define count 20000)

(define (random-value) (random-integer 100))
(define v (make-vector count #f))
(define (init)
  (let loop ((i 0))
    (if (< i count)
        (begin
          (vector-set! v i (make-point (random-value) (random-value)))
          (loop (+ i 1))))))
(begin (init) (##gc))

(define (distance x y)
    (sqrt (+ (* x x) (* y y))))
(define (swap a b)
  (let ((tx (point-x a))
        (ty (point-y a)))
    (point-x-set! a (point-x b))
    (point-y-set! a (point-y b))
    (point-x-set! b tx)
    (point-y-set! b ty)))
(define (compare a b)
    (let ((da (distance (point-x a) (point-y a)))
        (db (distance (point-x b) (point-y b))))
      (if (< da db)
        (begin (swap a b) #t)
        #f)))

(define (sort-v v)
  (let ((iMin 0) (n count))
    (let loop1 ((j 0))
      (if (< j (- n 1))
          (begin
            (set! iMin j)
            (let loop2 ((i (+ j 1)))
              (if (< i n)
                  (begin
                    (let ((comp (compare (vector-ref v i)
                                         (vector-ref v iMin))))
                      (if comp (set! iMin i)))
                    (loop2 (+ i 1)))))
            (if (not (= iMin j))
                (swap (vector-ref v iMin) (vector-ref v j)))
            (loop1 (+ j 1)))))))

(define (run) (sort-v v))
(define stats (##exec-stats run))
```

ce qui indique qu'un gain appréciable de performances est possible pour des programmes faisant beaucoup d'allocations.

La représentation plus compacte des enregistrements à travers l'utilisation de champs typés permet de réduire substantiellement le nombre de collections requises pour allouer un ensemble d'enregistrements. Déjà pour un point à deux dimensions, sauver un seul champ dans la représentation en compactant les deux dans un seul mot de 64 bits permet de sauver 11% du temps d'exécution total. Dans le meilleur cas où l'on a des enregistrements de 20 champs et des données entières de 8 bits, on économise 81% du temps d'exécution en utilisant la taille du tas par défaut. Augmenter celle-ci pour limiter le nombre de collections (à une plutôt qu'à six) permet de regagner un peu de terrain jusqu'à un ratio de 30%, car l'espace supplémentaire n'a aucun effet sur l'allocation des enregistrements plus compacts qui ne nécessitaient qu'une seule collection dans les deux cas.

Le ratio des temps de collections et d'exécutions du programme *##structure-20-big-heap* suggère que les performances du *garbage collector* ne diffèrent pas substantiellement entre les enregistrements typés et non typés malgré les modifications apportées. En effet, le temps de collections représente une proportion semblable du temps total d'exécution dans les deux programmes. Rappelons que le *garbage collector* doit extraire le descripteur de type et appeler *mark_array* individuellement sur chaque champ inclus dans la liste des champs *SCMOBJ*. Dans le cas d'enregistrements entièrement typés, on ne parcourt que le descripteur et on peut sauter par dessus tous les champs numériques. On observe aussi qu'avec le tas par défaut, la différence va dans l'autre sens, mais la proportion relative du temps utilisé par le *garbage collector* reste très proche pour les deux programmes.

Les primitives d'accès comportent, comme on s'y attendrait, un coût supplémentaire par rapport à la fonction *##unchecked-structure-ref*. Celle-ci se traduit par un appel direct à la macro *VECTORREF* (figure 5.7). On constate un surcoût d'environ 3% qui est probablement presque entièrement dû à l'utilisation de *BODY* plutôt que de *BODY_AS* qui utilise un masque. Comme on doit faire une opération de décalage, d'un et logique bits à bits et une négation versus une simple soustraction et que ces étapes supplémentaires représentent presque autant de travail que l'opération d'accès et qu'on n'économise qu'une opération de soustraction (qui se fait en une seule instruction) dans le calcul du

Figure 5.7 – *VECTOREF* vs *STRUCTREF*

```
#define ___UNTAG(obj)___CAST(___WORD*,(obj)&-(1<<___TB))

#define ___UNTAG_AS(obj,tag)___CAST(___WORD*,(obj)-(tag))

#define ___BODY(obj)(___UNTAG(obj)+___BODY_OFS)

#define ___BODY_AS(obj,tag) \
    (___UNTAG_AS(obj,tag)+___BODY_OFS)

#define ___VECTORREF(x,y) \
    *(___WORD*)(((___WORD)___BODY_AS(x,___tSUBTYPED))+ \
               ((y)<<(___LWS-___TB)))

#define ___STRUCTREF(x,y32,y64,type,proc) \
    *((___WORD *) (((___WORD)___BODY(x))+((y64)>>___TB)))
```

décalage requis, il est assez surprenant que le surcoût soit aussi bas.

De même, on observe une différence d'environ 5% entre les coûts des mutations pour des enregistrements normaux et des enregistrements typés. Encore une fois, la seule différence significative est l'utilisation de *BODY* versus *BODY_AS* (figure 5.8).

Le surcoût pour accéder à des champs entiers de 8 bits dans un enregistrement typé est de 6% par rapport à un champ analogue dans un vecteur d'entiers. Encore une fois, les deux opérations diffèrent surtout par l'utilisation de *BODY* (figure 5.9).

Les résultats des programmes un peu plus complexes sont présentés dans le tableau 5.II. Dans le cas de *distance sort*, on constate que les gains d'espaces assez minimes ne sont pas suffisants pour compenser les coûts supplémentaires aux opérations de *structure-*

Figure 5.8 – *VECTORSET* vs *STRUCTSET*

```
#define ___VECTORSET(x,y,z) \
    *(___WORD*)(((___WORD)___BODY_AS(x,___tSUBTYPED))+ \
               ((y)<<(___LWS-___TB)))=z;

#define ___STRUCTSET(x,z,y32,y64,type,proc) \
    *((___WORD*) (((___WORD)___BODY(x))+ \
                 ((y64)>>___TB)))=z;
```

Figure 5.9 – *S8VECTORREF* vs *INT8STRUCTREF*

```
#define ___FETCH_S8(base,i)*(___CAST(___S8*,base)+(i))

#define ___S8VECTORREF(x,y) \
    ___FIX(___FETCH_S8( \
      ___BODY_AS(x,___tSUBTYPED),(y)>>___TB))

#define ___INT8STRUCTREF(x,y32,y64,type,proc) \
    ___FIX(*((___S8*) \
      (((___WORD)___BODY(x))+((y64)>>___TB))))
```

Tableau 5.II – Résultats avec des enregistrements typés

Typed structures primitives			
name	baseline (s)	typed (s)	(ratio)
distance sort int32	1.54	1.76	(**1.14**)
distance sort int8	1.54	1.67	(**1.09**)
convex envelope int32	5.88	6.09	(**1.04**)
convex envelope int8	5.88	5.35	(**.91**)

ref et *set!* pour des enregistrements contenant des champs entiers de 32 bits ou 8 bits. La gestion des dépassements de capacité (*overflow*) dans les entiers de 32 bits cause un supplément de travail non négligeable par rapport aux entiers de 8 bits, avec un ratio de 114% versus 109%.

Les deux arrivent sans problème à compacter les deux champs dans un seul mot machine. La différence se trouve dans la distinction entre *INT* et *S32UNBOX* (figure 5.10) ainsi que l'opération inverse.

Les calculs d'enveloppe convexe font un peu mieux, arrivant à obtenir un léger gain

Figure 5.10 – *S32Unbox*

```
#define ___S32UNBOX(x) \
(___TYP((___temp=x)) == ___tFIXNUM \
 ? ___INT(___temp) \
 : ___BIGAFETCHSIGNED(___BODY_AS(___temp,___tSUBTYPED),0))
```

avec des *int8*. Dans les deux cas, le gain d'espace au moment de l'allocation devrait être le même puisque les champs des points bidimensionnels prennent au plus un mot machine par point. La différence de temps d'exécution entre 32 et 8 bits est beaucoup plus grand dans ce cas que pour *distance sort*, probablement parce que le ratio de références mutations est beaucoup plus favorable au tri, qui doit faire une grande quantité de références.

5.4 Vecteurs d'enregistrements

5.4.1 Programmes de tests

L'ensemble de programmes de tests servant à évaluer les performances relatives des vecteurs d'enregistrements va principalement utiliser comme comparatif des vecteurs contenant des références à des enregistrements. Les propriétés testées sont encore une fois l'accès, les mutations et les allocations. On teste les accès aux champs des vecteurs d'enregistrements, aux champs des références internes et aux descripteurs de type.

Dans le cas du programme testant l'allocation (figure 5.11), on a l'allocation d'un vecteur de 10 millions d'éléments puis on itère sur chacun des indices en assignant à chacun une référence vers un point nouvellement créé. On déclenche le *garbage collector* avec la fonction *##gc* avant de lancer l'exécution. Encore une fois, les statistiques d'exécutions sont collectées avec *##exec-stats*.

Son contrepoint utilisant un vecteur d'enregistrement (figure 5.12) va quant à lui allouer un vecteur d'enregistrements avec l'initialisation de l'arbre de typage que cela implique dans la fonction *run* puis écrire les valeurs 11 et 22 dans les champs de la référence interne appropriée en utilisant la fonction *point-vector-set!*.

Dans le cas des autres programmes, comme on n'essaie pas de tester si le temps d'allocation de l'arbre de typage et du vecteur d'enregistrement est compétitif, on préalloue le point auquel on souhaite accéder. Pour les tests d'accès au descripteur de types (figure 5.13), on itère de façon semblable aux boucles précédentes en propageant à chaque fois la valeur du descripteur de type recueilli par *##structure-type*.

Pour la version contenue du programme (figure 5.14), on alloue un vecteur de points et on conserve une référence (pas la première, qui, elle, a le descripteur de type à l'endroit

Figure 5.11 – *##structure* avec un vecteur

```
(declare (standard-bindings) (extended-bindings) (fixnum)
         (not safe) (block) (inlining-limit 0))

(define-type point id: point macros: x y)

(define count 10000000)

(define (run)

  (let ((v (make-vector count #f)))

    (let loop ((i (- count 1)) (result #f))

      (if (>= i 0)
          (begin
            (vector-set! v i (make-point 11 22))
            (loop (- i 1) v))
          v))))

(##gc)

(define s (##exec-stats run))
```

Figure 5.12 – *##structure* avec un vecteur d'enregistrements

```
(declare (standard-bindings) (extended-bindings) (fixnum)
         (not safe) (block) (inlining-limit 0))

(define-type point id: point macros: x y)

(define count 10000000)

(define (run)

  (let ((v (make-point-vector count)))

    (let loop ((i (- count 1)) (result #f))

      (if (>= i 0)
          (begin
            (point-vector-set! v i 11 22)
            (loop (- i 1) v))
          v))))

(##gc)
(define s (##exec-stats run))
```

Figure 5.13 – *##structure-type* avec vecteur

```
(declare (standard-bindings) (extended-bindings) (fixnum)
         (not safe) (block) (inlining-limit 0))

(define-type point id: point macros: x y)

(define stop 1000000000)

(define s (make-point 11 22))

(define (run)

  (let loop ((i (- stop 1)) (result #f))

    (if (>= i 0)
        (loop (- i 1) (##structure-type s))
        result)))

(define s (##exec-stats run))
```

Figure 5.14 – *##structure-type* contenu

```
(declare (standard-bindings) (extended-bindings) (fixnum)
         (not safe) (block) (inlining-limit 0))

(define-type point id: point macros: x y)

(define stop 1000000000)

(define c (make-point-vector 10))

(define s (point-vector-ref c 1))

(define (run)

  (let loop ((i (- stop 1)) (result #f))
    (if (>= i 0)
        (loop (- i 1) (##structure-type s))
        result)))

(define s (##exec-stats run))
```

usuel) vers l'un des enregistrements contenus. La primitive *##structure-type* a été modifiée pour parcourir l'arbre de typage lorsqu'elle rencontre une référence interne.

On réutilise aussi une version légèrement modifiée de *convex envelope* utilisant des vecteurs d'enregistrements plutôt que des vecteurs normaux. On fait la même chose pour distance sort. Dans les deux cas, on compare avec le même programme que dans les tests des enregistrements typés. On fait une substitution des appels à *make-vector* par *make-point-vector* et les *vector-ref* et *vector-set!* par *point-vector-ref* et *point-vector-set!* respectivement.

L'alternative la plus simple à l'utilisation de vecteurs d'enregistrements est d'allouer un vecteur contenant *NM* cellules où *N* est la taille de l'un des enregistrements et *M* est le nombre de ces enregistrements. Dans ce cas, on obtient une consommation mémoire analogue aux vecteurs d'enregistrements, mais l'information de typage est perdue et les accès aux cellules se font directement avec les primitives de vecteurs ce qui oblige de passer par une référence à l'origine du vecteur à chaque accès. Ces versions sont dénotées dans la colonne vecteur dans le tableau des résultats.

5.4.2 Résultats

Les résultats (tableau 5.III) varient entre des ratios de 2% et 451% pour l'allocation et les accès aux types en 64 bits respectivement. Dans le cas des allocations de vecteurs d'enregistrements, le temps d'exécution moyen est de 1.42 seconde en 64 bits et près d'une seconde est prise à allouer et initialiser l'arbre de typage puis à ajouter le vecteur dedans. Le reste de l'allocation (celle du vecteur lui-même) est presque instantanée même pour de grands vecteurs. Seulement une fraction de seconde est nécessaire pour initialiser les 10 millions de champs. L'allocation dans un vecteur normal évite ces coûts d'initialisation, mais passe beaucoup plus de temps dans l'allocation et la collection. Dans le cas d'allocation de grandes quantités de petites structures, plus d'une minute est passée dans le *gc*. L'utilisation de vecteurs de données contigues et non-typées offre des performances très comparable aux vecteurs d'enregistrements (environ le double du temps en 64 bits). La principale différence de performance semble être au niveau de l'accès aux cellules pour initialiser les valeurs. Les vecteurs ne supportant pas des références internes, on doit déréférencer le pointeur vers la structure englobante pour accéder aux cellules à chaque écriture alors que l'accès ne fait qu'incrémenter un pointeur pour atteindre chaque champ dans la structure dans le cas de vecteurs d'enregistrements.

Les accès aux enregistrements individuels d'un vecteur d'enregistrements sont passablement plus lents que pour un vecteur de références vers des enregistrements, avec des ratios de 1.23 en 32 bits et 1.48 en 64 bits respectivement.

En comparant les deux opérations, on constate que la différence majeure est que l'accès dans un vecteur d'enregistrements, fait avec *CONTAINERREF* (figure 5.15), évite une indirection de pointeur, retournant le pointeur directement après lui avoir ajouté le

Figure 5.15 – *CONTAINERREF*

```
#define ___CONTAINERREF(c,s,i) \
    (___SCMOBJ) ___TAG(((___WORD*) \
         ___UNTAG_AS(c,___tSUBTYPED))+ \
        (___INT(i) * ___INT(s)), ___tMEM2)
```

	Base		Vecteur d'enregistrements				Vecteur			
	32	64	32		64		32		64	
structure alloc	78.47	72.01	1.26	(**.02**)	1.42	(**.02**)	1.10	(**.01**)	3.07	(**.04**)
structure20 alloc	5.00	5.17	.35	(**.07**)	.65	(**.12**)	1.46	(**.29**)	1.40	(**.27**)
structure20 alloc gc	4.45	4.55	.00	(**.00**)	.00	(**.00**)	.00	(**.00**)	.00	(**.00**)
structure20 alloc no-init	4.85	5.23	.34	(**.07**)	.57	(**.11**)	1.46	(**.30**)	1.40	(**.27**)
structure access	1.32	1.24	1.63	(**1.23**)	1.84	(**1.48**)	.95	(**.72**)	.63	(**.51**)
structure set !	4.64	4.19	1.24	(**.27**)	1.24	(**.30**)	1.13	(**.24**)	2.14	(**.51**)
type access	2.07	1.85	7.42	(**3.59**)	8.35	(**4.51**)				
prop-access	1.33	.93	1.19	(**.90**)	.94	(**1.01**)	.95	(**.71**)	.63	(**.68**)
convex envelope	.87	.62	.82	(**.93**)	.55	(**.89**)				
distance sort	3.42	3.52	2.96	(**.87**)	3.00	(**.85**)				
quicksort	1.36	.81	1.20	(**.89**)	.85	(**1.05**)				
convex envelope safe	6.91	5.50	7.66	(**1.11**)	5.95	(**1.08**)				
quicksort safe	2.10	1.71	3.67	(**1.75**)	3.58	(**2.09**)				

Tableau 5.III – Résultats avec des vecteurs d'enregistrements

bon décalage. Plusieurs opérations arithmétiques (*TAG* est une addition, *INT* un *shift* vers la droite) sont toutefois requises pour calculer ce dernier. Ici les vecteurs simples sont substantiellement plus rapides (presque par un facteur 3 en 64 bits), car on accède à une seule des cellules de l'enregistrement à chaque fois ce qui réduit l'avantage des vecteurs d'enregistrements.

La mutation des enregistrements est près de quatre fois plus rapide sur des enregistrements contenus. Dans ce cas, la version sur vecteur récupère chaque enregistrement et appelle la méthode de mutation sur chacun de ces champs. La version contenue va quant à elle utiliser la primitive *##structure-vector-set !* à travers la fonction *point-vector-set !*. Comme toutes les cellules sont adjacentes, l'accès est très efficace, car la méthode de mutation va faire l'écriture en bloc. On retrouve donc ici aussi le même avantage à regrouper les accès par rapport aux vecteurs.

Le pire résultat est, comme on s'y attendrait, l'accès au descripteur de type, qui prend 3.59 à 4.51 fois plus longtemps dans les enregistrements contenus. La nécessité de faire une recherche dans un arbre à partir de la valeur de la référence augmente de beaucoup le coût par rapport à un simple accès au premier champ de l'enregistrement. On effectue cinq indirections et plusieurs opérations arithmétiques pour récupérer le descripteur de

type.

L'accès aux champs d'un enregistrement (le programme *prop-access*) se fait avec la même méthode dans les deux cas et le fait que l'une d'elles soit contenue n'a aucun effet sur le temps requis. Les vecteurs sont, eux, légèrement plus rapides, car la référence pointe directement sur la cellule (en fait le code est exactement le même que pour l'accès à un enregistrement) et non sur son descripteur de type (ou à l'endroit où il devrait être).

Compacter les descripteurs au début d'un vecteur d'enregistrements pourrait occasionner des coûts de performances prohibitifs dans un programme faisant un usage intensif des types, mais nous supposons que, dans une utilisation typique, les gains au niveau de l'allocation compensent ce surcoût. En effet, dans les programmes optimisés, les tests de typages sont déjà éliminés partout où ils ne sont pas strictement nécessaires.

On obtient une amélioration majeure dans le programme d'enveloppe convexe qui fait un usage intensif de *structure-set!* et l'allocation de nombreux enregistrements qui rendent les gains au niveau du *garbage collector* substantiels (4 *gc* contre aucun). Le programme *distance sort* n'obtient quant à lui qu'un gain mineur. Le peu d'allocations ne donne pas d'avantage notable à compacter les données, et la nécessité de copier les valeurs dans des variables temporaires avant de faire la permutation limite les opportunités de profiter de *point-vector-set!*.

5.5 Combinaisons

5.5.1 Programmes de tests

Pour tester la combinaison de vecteurs d'enregistrements et d'enregistrement contenant des champs typés, on réutilise les mêmes programmes que dans la section précédente.

Le seul changement effectué est d'ajouter des annotations de types aux champs (figure 5.16). On teste avec des champs de 8 et 32 bits sauf pour l'accès aux références internes (*structure access typed*) puisqu'il n'y aurait aucune différence de représentation mémoire entre les deux et que l'opération est indépendante du type des champs. La version *baseline* est celle utilisant un vecteur de références vers des enregistrements normaux.

On a donc un test d'accès aux enregistrements internes versus une référence externe

Figure 5.16 – Conversion vers les programmes typés

```
(define-type point id: point macros: x y)

    =>

(define-type point id: point macros: (x int8:)  (y int8:))
(define-type point id: point macros: (x int32:) (y int32:))
```

(*structure-access-typed*), des tests de mutations de champs entiers de 8 et 32 bits (*structure-set !*), l'allocation de 10 millions d'enregistrements typés dans un conteneur (*structure-alloc*) et un programme calculant l'enveloppe convexe d'un nuage de points (*convex envelope*).

5.5.2 Résultats

Dans la table de résultats 5.IV, la colonne ratio correspond au ratio entre le temps d'exécution du programme utilisant les vecteurs d'enregistrement (*contained*) sur le temps d'exécution du programme normal (*baseline*). La colonne *untyped* recopie simplement les résultats de la section précédente.

Tableau 5.IV – Résultats avec des vecteurs d'enregistrements typés

Typed Structure Vectors					
name	baseline (s)	contained (s)	(ratio)	untyped (s)	(ratio)
structure access typed	2.47	2.50	(**1.01**)	2.50	(1.01)
structure set ! int8	4.45	2.28	(**.51**)	2.28	(.51)
structure set ! int32	4.45	2.30	(**.52**)	2.28	(.51)
structure alloc int8	2.90	.86	(**.30**)	1.02	(.35)
structure alloc int32	2.90	1.66	(**.57**)	1.02	(.35)
convex envelope int8	5.87	1.25	(**.21**)	1.26	(.21)
convex envelope int32	5.87	1.29	(**.22**)	1.26	(.21)
distance sort int8	1.54	1.64	(**1.07**)	1.51	(.98)
distance sort int32	1.54	1.74	(**1.13**)	1.51	(.98)

Les accès et mutations sur des références internes à des vecteurs d'enregistrements restent essentiellement inchangés indépendamment du type, comme on pourrait s'y

attendre, puisque les opérations sont presque identiques. Le léger surcoût associé aux *int32* refait encore une fois surface, mais est ici très mineur à moins d'un pour cent.

Dans le cas de l'allocation d'enregistrements, on a un gain considérable ($\sim 20\%$) en utilisant des *int8*. Le temps d'exécution est ici inférieur à une seconde, car on conserve le même nombre d'objets alloués que dans les autres versions précédentes. Ici, les *int32* sont plus lents de 56% que la version non typée avec des entiers *boxés* d'une précision de 30 bits.

Les *benchmarks* d'enveloppes convexes donnent environ les mêmes résultats avec des champs typés et non typés. Les *int32* sont légèrement plus lents alors que les *int8* et les *SCMOBJ* ont des temps d'exécution essentiellement identiques à une fraction de seconde près.

On constate que pour le tri par distance, l'ajout de champs typés représente un surcoût important que ce soit en vecteurs d'enregistrement ou pas. Les deux versions typées sont significativement plus lentes que les versions non typées (9 et 15% respectivement pour 8 et 32 bits).

5.6 Travaux connexes

De nombreux travaux ont été faits sur la gestion de mémoire des langages de programmation. Essentiellement tous les langages typés statiquement offrent la possibilité de spécifier des types hétérogènes dans des enregistrements puisque les méthodes sont spécialisées au moment de la compilation grâce à une analyse de type. Des travaux ont été faits par Keep et Dybvig [**?**] sur *Chez Scheme* pour introduire un analogue aux *struct* C.

L'idée de regrouper des objets partageant certaines propriétés ensemble pour simplifier la logique d'allocation et de libération n'est évidemment pas nouvelle. La gestion de mémoire basée sur des régions[**?**] telle que présentée par Tofte et Talpin ([**?**]) dans SML est généralement considérée comme la forme canonique de cette approche. Une approche couramment utilisée qui présente plusieurs similarités avec les vecteurs d'enregristrements est celle du *Big Bag of Pages* qui dédie des pages de mémoire pour des types spécifiques. De même, certains algorithmes de *gc* (comme Boehm, par exemple [**?** ,

p. 166]) comportent une gestion des références internes semblable à celle des vecteurs d'enregistrements.

L'inspiration pour l'arbre de typage vient d'un algorithme classique largement utilisé dans les systèmes d'exploitation pour optimiser la pagination de la mémoire [**? ?**].

5.6.1 Chez Scheme

Keep et Dybvig ont publié en 2010 une extension à Chez Scheme ajoutant des *foreign type structures* permettant de déclarer des types avec une macro *define-ftype* (exemple dans la figure 5.17) qui contiennent des donnés C externes. Ils utilisent une interface de génération basée sur la macro *define-type* qui est semblable à celle décrite ici (voir figure).

Leur implémentation permet n'importe quelle forme de données externes comme des pointeurs de fonctions, des unions, des chaînes de bits, etc., en plus de pouvoir spécifier l'*endianness* et des champs de remplissage, mais ne permet toutefois pas de combiner les données externes avec les données Scheme. Toutes les opérations sur des *ftypes* utilisent des fonctions dédiées plutôt que d'utiliser la même interface que les enregistrements normaux. Il s'agit donc plutôt d'une version étendue d'une interface pour objets externes que d'une façon d'amener des objets typés à Scheme.

5.6.2 Allocation par régions

La gestion de mémoire par régions telle que présentée par Tofte et Talpin comprend certaines similarités avec le concept de vecteur d'enregistrements, notamment les algorithmes d'inférences de régions et l'idée qu'une région est un bloc monolithique alloué ou libéré comme un seul objet et demeurant vivant tant qu'il existe des références vers celui-ci. Leur implémentation repose sur une sémantique statique étendue et un système de type inféré par unification permettant de prendre en compte les effets de bord sur les régions.

Leur approche diffère en allouant tous les objets dans des régions et en omettant un *garbage collector*. L'analyse de vivacité des différentes régions est faite statiquement et celles-ci sont libérées par des appels au désallocateur. Comme aucun test de typages

Figure 5.17 – Utilisation de *define-ftype*

```
(define-ftype x-type
  (struct
    [a (union
         [s (array 4 integer-16)]
         [i (endian big (array 2 integer-32))]
         [d (endian little double)])]
    [b (bits
         [x signed 4]
         [y unsigned 4]
         [z signed 3]
         [_  signed 5])]
    [c (packed
         (struct
           [ch char]
           [_  integer-8]
           [us unsigned-16]))]
    [f (* fun-type)]
    [next (* x-type)]))
```

n'est requis à l'exécution, les descripteurs de types et les entêtes peuvent être omis sans problème de la représentation des objets.

5.6.3 BIBOP

Certains allocateurs, notament dans *Chez Scheme*, utilisent des *big bad of pages* pour optimiser la représentation de types. Cette approche est très similaire à celle utilisée pour les vecteurs d'enregistrements. En effet, on utilise des pages de mémoire dédiée à un type spécifique et une table pour les indexer, ce qui évite d'avoir à stocker les types directement sur les valeurs immédiates. Cette approche est normalement utilisée pour optimiser la représentation de types prédéfinis et des listes de segments de mémoire et des pointeurs d'allocations distincts doivent être maintenus pour chaque type supporté. Notre approche, elle, ne supporte que des enregistrements, mais ne requiert pas de mécanique d'allocation séparée pour chaque type.

5.6.4 Table de pagination multiniveau

L'algorithme utilisé pour récupérer les descripteurs de types à partir d'une référence interne à un vecteur d'enregistrement est fortement basé sur les tables de paginations multiniveaux. Celles-ci existent de longue date et sont largement utilisées dans le domaine des systèmes d'exploitation. Cette approche a été proposée en 1975 par Greenberg et Webber.

Notre algorithme comprend quelques modifications pour l'appliquer à une utilisation où l'ensemble de la mémoire n'est pas nécessairement utilisé (contrairement à un gestionnaire de mémoire au niveau du système d'exploitation) et dont la majorité des pages n'est en général pas indexée dans l'arbre. Pour ce faire, on introduit le bloc nul et des méthodes servant à ajouter des branches ou à élaguer dynamiquement. De même, les feuilles ne réfèrent pas obligatoirement à un objet occupant une seule page, donc un soin particulier doit être apporté à la gestion des tables.

5.6.5 GC de Boehm

Certains algorithmes de *gc* non précis (Boehm par exemple) permettent de gérer les pointeurs à l'intérieur d'un objet comme des références vers l'objet lui-même ce qui est semblable à la gestion mémoire des vecteurs d'enregistrements. Une stratégie typique de ce genre de *gc* conservateur est d'allouer des blocks de mémoire seulement pour des objets de même taille, ce qui permet de retrouver le début de l'objet à parcourir à l'aide de son alignement. Cette stratégie est semblable à l'arbre de typage utilisé par les vecteurs d'enregistrements, mais requiert généralement que chacun des champs soit parcouru dans un objet puisque l'information de typage n'est pas disponible à l'exécution.

CHAPITRE 6

CONCLUSION

Afin d'optimiser la gestion de la mémoire des enregistrements dans le compilateur Gambit Scheme, nous avons introduit des annotations de types sur les champs et ajouté des primitives permettant l'utilisation de vecteurs d'enregistrements.

6.1 Enregistrements typés

Ces annotations de types sont associées à des champs contenant des valeurs entières ou flottantes à précision fixe entre 8 bits et 64 bits et donnent au programmeur un meilleur contrôle sur la représentation en mémoire des enregistrements ce qui permet des gains d'espaces allant jusqu'à 8x (entiers 8 bits sur une architecture 64 bits).

Leur implémentation demande l'introduction de primitives d'accès et de mutation spécialisée ainsi que des modifications substantielles à la primitive d'allocation et au *garbage collector*. En effet, la présence de champs non *boxés* à travers les objets Scheme typique impose que la position des champs dans l'enregistrement ne soit pas alignée sur un multiple d'un mot machine et varie donc en fonction de l'architecture utilisée. Ces champs vont aussi poser problème au *gc*, car ils ne contiennent pas les bits d'annotation de typage retrouvés sur les autres objets Scheme. On doit donc avoir un moyen de sauter par dessus lors du parcours de l'enregistrement. On règle ces difficultés en propageant les positions relatives des champs pour des architectures de 32 bits et 64 bits dans le descripteur de type (pour le *gc*) et dans la construction des appels aux primitives. On doit aussi modifier le *gc* pour récupérer le descripteur de type de l'enregistrement et parcourir l'ensemble des décalages pouvant contenir des références.

Ces ajouts permettent donc au programmeur de spécifier plus précisément la représentation des enregistrements et d'obtenir des gains d'espaces conséquents au coût d'un surplus de travail qui peut être majoritairement fait au moment de la compilation.

6.2 Vecteurs d'enregistrements

La représentation des enregistrements requiert un mot d'entête et un mot contenant une référence vers le descripteur de type le décrivant. Il s'agit d'un surcoût considérable pour de petits objets. En effet, pour allouer M enregistrements de N champs normaux, on a besoin de $M(2+N)$ mots machines. Nous avons réduit ce surcoût en regroupant les enregistrements de mêmes types dans des vecteurs plats d'enregistrements. Ceux-ci permettent d'utiliser un seul entête pour tout le vecteur et de conserver uniquement une instance de la référence vers le descripteur de type qui est partagé entre tous les enregistrements. Cette méthode d'allocation permet d'allouer les mêmes enregistrements en seulement $2+MN$ mots. Dans le meilleur cas, on utilise jusqu'à 10 fois moins d'espace que l'implémentation existante : pour un grand nombre de structures contenant 8 entiers de 8 bits, on utilise $2+M$ versus $(2+8)M$.

Comme Scheme est un langage typé dynamiquement, on ne peut pas effacer les types. Le fait de condenser toutes les références au descripteur dans un seul champ rend difficile son accès puisqu'il peut se trouver à n'importe quel décalage par rapport à la référence interne. Pour régler ce problème, on utilise un arbre de typage multiniveau inspiré des tables de pagination de mémoire. Celui-ci comprend une profondeur constante ce qui permet de faire un accès inconditionnel au descripteur de type à partir de l'adresse de la référence. Pour limiter l'utilisation de mémoire de l'arbre, on condense tous les chemins non utilisés vers un bloc nul autoréférent. Des branches sont ajoutées ou élaguées de l'arbre lors de l'allocation ou de la libération de vecteurs d'enregistrements.

L'implémentation des vecteurs d'enregistrement et de l'arbre de typage associé requiert d'autres modifications au *runtime*. On doit ajouter des primitives d'allocation, d'accès et de mutation et modifier les tests de types pour gérer correctement les enregistrements contenus n'ayant pas d'entête propre. Le tout permet d'utiliser les références internes de façon complètement transparente avec des enregistrements normaux.

Il est aussi nécessaire de modifier le *garbage collector* pour prendre en compte les références internes qui doivent garder l'ensemble du vecteur en vie tant que l'une d'elles subsiste et l'élagage de l'arbre de typage lorsqu'il est désalloué.

Ces allocations en masses permettent à la fois de mieux contrôler les collections et de réduire les surcoûts associés à la représentation des enregistrements. Ils nécessitent néanmoins un travail supplémentaire considérable au moment de l'allocation pour construire et maintenir l'arbre de typage. Elles entraînent aussi des changements légers, mais coûteux aux primitives accédant aux structures qui doivent maintenant supporter deux marqueurs de type sur les pointeurs pour différencier entre des références normales et internes ce qui complexifie l'extraction du corps de l'enregistrement.

6.3 Résultats

Pour évaluer l'impact sur la performance des changements apportés, nous avons utilisé une série de petits programmes de tests en séparant l'utilisation de champs typés, de vecteurs d'enregistrements et d'une combinaison des deux. Les trois ensembles sont comparés à une implémentation équivalente utilisant les primitives existantes.

En comparant des enregistrements typés et non typés, on observe un coût supplémentaire de 3 à 6% pour les accès et les mutations. Ce coût vient de la nécessité d'utiliser un masque plutôt qu'une soustraction pour enlever les bits de *tags* des références et pourrait être omis sans les vecteurs de structures. On observe aussi des gains allant de 11% à 81% pour l'allocation d'enregistrements en fonction du nombre et de la taille des champs.

Deux programmes plus complexes montrent pour le calcul d'enveloppe convexe une légère perte de performance sauf avec des entiers de moins de 32 bits où les gains à l'allocation compensent le surcoût à l'accès. Un tri naïf donne des pertes de 9% à 14% en faisant un faible nombre d'allocations et beaucoup de comparaisons. En général, les coûts supplémentaires semblent être acceptables et les gains lors de l'allocation devraient pouvoir les compenser dans des programmes ayant des goulots d'allocation.

Les vecteurs d'enregistrements, quant à eux, obtiennent des gains sur presque tous les programmes de tests. Les accès aux références internes et à leurs propriétés se font essentiellement dans le même temps que pour des enregistrements normaux. Les autres opérations sont entre 1.96 et 4.76 fois plus rapides à l'exception des tests de typages. Ceux-ci sont 3.73 fois plus lents puisqu'ils doivent parcourir l'arbre de typage pour

récupérer le descripteur de type.

L'ajout des champs typés aux enregistrements contenus ne donne pas de gains significatifs par rapport aux vecteurs d'enregistrements sans ceux-ci. Ils permettent une représentation plus compacte, mais l'allocation en masse limite déjà efficacement le déclenchement du *garbage collector*. Le coût supplémentaire associé au *unboxing* des entiers de plus grande taille nuit même en général légèrement aux performances. Les entiers 8 bits donnent néanmoins un léger gain (1.23 fois plus rapide) au niveau de l'allocation.

En général, l'ajout de vecteurs d'enregistrement semble gagnant pour des programmes faisant beaucoup d'allocations et ne nécessitant pas de nombreux tests de typages. Ceux-ci peuvent habituellement être éliminés des parties du code ayant des contraintes de performance sauf s'ils sont explicitement requis. Les enregistrements typés démontrent quant à eux des gains seulement au niveau de l'allocation d'enregistrements, et ces gains semblent limités lorsqu'on les combine avec des vecteurs d'enregistrements. Compacter la représentation en introduisant des champs typés semble plus approprié pour des applications allouant (ou copiant) des enregistrements de façon très intensive.