Structure Vectors and their Implementation

Benjamin Cérat
Université de Montréal
ceratben@iro.umontreal.ca

Marc Feeley
Université de Montréal
feeley@iro.umontreal.ca

Abstract

The typical representation of structures (a.k.a. records) includes a header and type descriptor that are a considerable overhead when the structures have few fields and the program allocates a large number of them. We propose *structure vectors* that group many structures of the same type, removing the need for a header and type descriptor for each contained structure. This paper describes our implementation of structure vectors within the Gambit Scheme system. Microbenchmarks indicate that structure allocation is faster, structure access is roughly the same speed, and type checking is substantially slower. On real applications we have observed speedups of 7% to 15%.

1. Introduction

Scheme structures (a.k.a. records) of f fields can be straightforwardly implemented as a specially tagged vector of length $f+1$ containing a reference to a type descriptor and the values of the fields. The type descriptor is useful to attribute to the structure a unique type different from all other types of structures. It is also a convenient place to store meta information such as the field names used for pretty-printing, and the super type in systems supporting structure type inheritance. In a typical memory management system, memory allocated objects are prefixed by a header containing the object's primary type (e.g. to distinguish vectors from structures), a length, and fields used by the garbage collector. For a structure, the space for this header and the type descriptor adds an overhead that can be relatively high when the number of fields is small.

In the Gambit Scheme system (a Scheme to C compiler), the header, type descriptor, and the fields, each occupy a machine word (32 or 64 bits). Moreover, small objects (less than 256 words) are managed using a Cheney-style compacting garbage collector with a factor of two memory use bloat due to the unused but reserved space in the tospace (i.e. an object of length n words causes $2n$ words of memory to be reserved for it). Given that structures are often small, there is a considerable space overhead and a run time overhead for managing structures. Figure 1 shows the layout of a 2D point structure with fields `x` and `y` and a tagged reference to the structure. Note that Gambit uses the two lower bits for the tag, and memory allocated objects are either tagged with `11` for pairs, or `01` for all other types). The state of the structure is stored in 4 words, but a total of 8 words of memory will be reserved for this structure by the memory manager (the tospace remains live between garbage collections to guarantee that a garbage collection can always be performed without running out of memory).

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

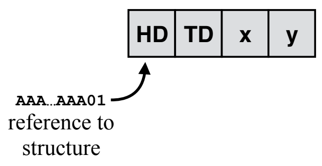

Figure 1: A 2D point structure type definition and its representation with a header, type descriptor and `x` and `y` fields

In some applications, it is necessary to manage many structures of the same type and there is a delimited region of code where these structures are live. An example is the construction of a 3D polygon-based model composed of 3D points to be sent to a GPU for rendering. The data becomes dead after it is sent to the GPU.

For such applications we propose *structure vectors*, a compact representation of a group of structures of the same type. The type descriptor is only stored once in the structure vector, and each contained structure occupies space for its fields only. Figure 2 shows the layout of a 3 element structure vector of 2D points. In typical uses, structure vectors are large objects so they are managed by a non-compacting garbage collector that does not suffer from the factor of two bloat of the copying garbage collector. The allocation of an n element structure vector of structures with f fields requires

Scheme '14, November 19, 2014, Washington, D.C., H Street NW, USA.
Copyright © 2014 ACM 978-1-nnnn-nnnn-n/14/11…$15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

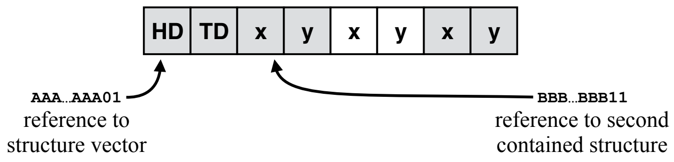

Figure 2: Representation of a structure vector containing 3 2D points

$2 + nf$ words. This compares well to the $2n(2 + f)$ words required for allocating the structures individually. For a large n there is a factor of $2 + 4/f$ space savings (e.g. a factor of 4 for $f = 2$, a factor of 3.333 for $f = 3$, a factor of 3 for $f = 4$).

There are a few important issues to address in implementing structure vectors. To allow handling each contained structure individually, e.g. passing them to a function expecting a structure, it is necessary to have internal references to the elements of the structure vector. Operations on structures (access to fields, type checking, etc) must work transparently on individually allocated structures and contained structures. To access the type descriptor of a contained structure, and implement the garbage collector, there must be a way to find the structure vector that contains the structure given a reference to that structure.

This is solved by storing all structure vectors in a single fixed depth *container tree*, similar to the page tables used in the implementation of virtual memory (Silberschatz et al. [1]), that maps addresses to the structure vectors that span that address. A single container tree is needed for managing all live structure vectors. The use of a fixed depth tree has the advantage that an address lookup can be done with a fast to execute fixed number of unconditional indirection.

In the next section, we cover the algorithms used to construct and maintain the container tree. In section 3, we present the modifications to the Gambit Scheme runtime required by the implementation of structure vectors. This is followed by a brief performance evaluation and a discussion of related work.

2. Container Tree

The container tree maps contained structure references to the structure vector that contains them. This tree spans the whole memory and is indexed using a fixed number of bit fields from the reference (Figure 3). Some of the lower bits of a reference are unused in the container tree indexing process. This is possible because of the alignment constraint imposed on structure vectors that are aligned to addresses that are multiples of 4096. The container tree is allocated in the C *heap* using *malloc* and only one instance is created for the *runtime*. It is accessed through the C functions that implement container allocations, etc. It is also accessible to the

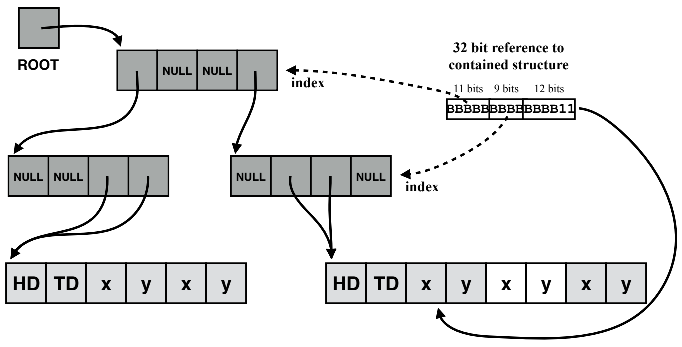

Figure 3: Container tree and access using a 32 bit reference to a contained structure

garbage collector that keeps it up to date during collection (but it is not a GC root).

The nodes at a given level of the tree are vectors of the same power of 2 size containing pointers to nodes at the next level of the tree, or to structure vectors at the leaves of the tree. Different layouts for the two supported word sizes (32 and 64 bits) are used to minimize the amount of memory used. Whenever a structure vector is allocated or freed, the tree is automatically updated to reflect those changes using runtime methods hooked to the allocator and garbage collector respectively. Accesses to the leaves of the container tree are performed by an unconditional chain of indirection calculated directly from the structure reference.

2.1 Managing Nodes

The container tree is composed of two levels on 32 bit word platforms and five levels on 64 bit platforms. A reference to the top level node is kept in a global variable (*ROOT*) so as to make it available to the *runtime*. Each node contains a field for every distinct value possible in the set of bits indexing it. The root node is slightly larger than the others since it is unique while at least one new node is usually allocated for typical (large) vector. References to the vectors are chained through the nodes with a pointer to the next node being left in the corresponding field. The leaves of the tree contain *boxed* pointers to the structure vectors.

In order to simplify testing for unused paths and preserve space while ensuring that every single path down the tree has the same length, we use the *NULL* pointer as a marker for absent subtrees. This allows testing if a path is used through a simple pointer comparison. We can guarantee through runtime type tests that only internal references will ever be followed in the container tree since only contained structures have the correct *tag*.

To add a structure vector to the tree, we first need to add nodes to the tree to ensure that the addresses spanned by the vector have corresponding full depth subtrees in the container tree. To do so, we recursively go down in the subtree

```
#if ___WORD_WIDTH == 32

#define ___GET_CONTAINER(ref) \
    (ROOT[(ref)>>21] \
         [((ref)>>12) & 0x1FF])

#else

#define ___GET_LOW_OFS(n) \
    (___WORD_WIDTH - (n*10) - 9)

#define ___GET_CONTAINER(ref) \
    (ROOT[(ref)>>52] \
      [((ref)>>(___GET_LOW_OFS(1))) & 0x3FF]\
      [((ref)>>(___GET_LOW_OFS(2))) & 0x3FF]\
      [((ref)>>(___GET_LOW_OFS(3))) & 0x3FF]\
      [((ref)>>(___GET_LOW_OFS(4))) & 0x3FF])

#endif
```

Figure 4: Container tree access macros

spanned by the vector adding a node each time we come across a NULL pointer. Once all the nodes have been added, we write references to the vector in every leaf that corresponds to the addresses it spans. Note that the allocation can be optimized to add at most 3 nodes to each level of the tree because the middle node can be reused (it is an array containing the same pointer).

Since structure vectors are managed by the garbage collector, we add a hook to ensure that all references to the vector are removed from the container tree before it is freed. As we always allocate the vector as a non-movable object, we don't have to worry about a change in the contained address (which would require rebuilding the whole tree) during the collection. To prune the now unneeded subtree, we recursively check whether the nodes are shared and then free them if they are not. By construction, only the outermost nodes at a given level may be shared because structure vectors span contiguous addresses so we can get away with only checking these. We make sure that all references that belong to the vector we are removing are set to NULL.

2.2 Accessing the Vectors

The container tree has a fixed depth for a given architecture and every path for a live contained structure is guaranteed to have that depth. This property allows us to navigate it to fetch the structure vector corresponding to an address without any conditional tests. We do this by systematically extracting the relevant bits for a given level to calculate the index at which the reference to the next level will be stored. The last such reference will be a *boxed* pointer to the vector.

The container tree is used by the garbage collector when it encounters a reference to a contained structure by allowing it to find the structure vector containing the contained structure. This structure vector is then considered live by the garbage collector.

```
(define-type foo x y z)
(define c (make-foo-vector 1000000))
(define s (foo-vector-ref c 999))
(foo-vector-set! c 999 11 22 33)
(foo-x s) ;; => 11
(foo-y-set! s 44)
(foo? s) ;; => #t
```

Figure 5: Example of structure vector functions

Scheme code also accesses the container tree when getting the type descriptor of a contained structure, using the function *##contained-type* that is directly inlined in the generated code.

3. Structure Vectors

Our implementation of structure vectors use the *define-type* (Figure 5) macro as an interface. The macro was extended to generate definitions for a structure vector constructor, and getter and setter specialized to the type. Those definitions (whether functions or macros) are constructed from the type information using a set of new primitives. Several changes to the Gambit *runtime* were made to introduce these, notably to the tagging scheme, to *define-type* and to the garbage collector.

3.1 Tagging

In order to maintain compatibility with existing accessors, we re-purpose the *tag* used by pairs, i.e. `11`, to dedicate it to contained structures (pairs now use the `01` tag and the header needs to be accessed by the `pair?` primitive). Consequently, a reference to a structure can be tagged with `01`, when it is an individually allocated structure, or with `11`, when it is a structure contained in a structure vector.

The type-checking primitives for structures must account for the two possible structure layouts. Given that there are now two different *tags* denoting structures, we must also switch our field access primitives from using a simple substraction (which the C compiler is normally able to optimize away) to a mask removing the *tag* bits when *unboxing* a reference (Figure 6). In other words, we must use *___UNTAG(obj)* rather than *___UNTAG_AS(obj,___tSUBTYPED)*. Since Gambit does not currently optimize redundant *boxing* and *unboxing*, these extra operations represent a significant overhead on structure accesses.

3.2 Structure Vector Primitives

For a structure type name *foo*, the *make-foo-vector*, *foo-vector-ref* and *foo-vector-set!* definitions are built as calls to primitives. The first allocates a large *non-movable* object as a structure vector and then adds it to the container tree. To ensure that no structure vector shares the same page, extra memory equal to the page size is allocated at the end of the object. This form of allocation is managed by a *mark and sweep* collector and is reserved for large objects (over 1

```
#define ___TB 2

#define ___tSUBTYPED   1
#define ___tCONTSTRUCT 3

#define ___TAG(ptr,tag) \
    (((___WORD)ptr)+(tag))

#define ___UNTAG(obj) \
    ((___WORD*)((obj)&-(1<<___TB)))

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

Figure 6: C macros to tag and untag references

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

Figure 7: Code generated to access a contained structure

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

Figure 8: Type access primitive

kilobytes). Fragmentation of this memory space is no worse then using C's *malloc* since only large memory blocks are allocated there.

The definition for *foo-vector-ref* is compiled to a C macro (Figure 7) that simply bumps the pointer to the vector and *retags* it to the contained structure *tag*. The offset is calculated by passing it the index of the structure and it's size (in words). The structure's size is provided by the *define-type* macro. In order to maintain full transparency when using regular structure functions on contained structures, the pointer returned by the accessor is offset by the usual amount from the first field and thus points two fields into the previous structure.

The macro or function (*foo-vector-set!*) supplements the normal constructor (*make-foo*). It initializes a structure in the vector by setting all of its fields to the values passed in parameters. The compiled code thus resembles closely the normal constructor without, of course, the allocation.

We have modified the primitives that deal with type testing (*##structure-type*, *##structure-instance-of?*, etc.) to differentiate internal references from normal structures (Figure 8) and to recover their type descriptor through the container tree instead of accessing the first field in the structure.

All the primitives provided except the allocator (where almost all the work is done directly in C anyway) are automatically inlined to C macros in code declared as *unsafe*. Type

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

Figure 9: Declaration used for the benchmarks

checks in those primitives are also automatically removed by specializing the calls to the unchecked version.

3.3 Changes to the Garbage Collector

The addition of structure vector primitives requires slight modifications to the garbage collector. First we need to ensure that the container tree is updated whenever a vector is reclaimed. To do so we introduce a new subtype for structure vectors (several values are still unused in our subtyping scheme so this does not pose any problem). Whenever we reclaim a *non-movable* object, we test to see if it matches this subtype and call a method to prune the tree as necessary. We also need to ensure that a vector is never freed while a reference to a structure it contain is still live. To do so, whenever we encounter a reference to a contained structure (with the *tag ___tCONTSTRUCT*), we recover the vector itself with *___GET_CONTAINER* and substitute it to the object being scanned.

4. Evaluation

To assess the performance of the new primitives, we use benchmarks that are implemented both using individually allocated structures (baseline) and with structure vectors. To remove outliers, we run each benchmarks 20 times and remove the highest and lowest value. We then take the geometric mean of the remaining values. We have set the various programs to have execution times of at least around 1 second. All the benchmarks were run in both 32 bits and 64 bits mode on a machine with a 2.2 GHz *Intel core i7* with 8 GB of RAM.

Our benchmark programs were compiled by using *gsc* to generate an executable file. To ensure similar execution between the baseline and structure vector versions, we use a set of declarations (Figure 9). The *standard-bindings* and *extended-bindings* declarations allow the compiler to assume that primitives are never redefined and can thus be inlined. The *fixnum* declaration allows the use of fixed precision integers instead of the generic numeric tower. *Not safe* lets the compiler specialize primitives into unsafe versions and perform other optimization. *Block* specifies that the whole program is contained in the file. The *inlining-limit* sets a maximum factor of growth that is acceptable during inlining. It is set to 0 (no growth) to prevent different loop unrolling between comparable benchmarks. To avoid making unnecessary function calls, we also specify that all type definitions generate their methods as macros.

4.1 Benchmarks

To evaluate important aspects of our system's performance, we have a series of benchmarks testing specific aspects: *structure alloc*, *structure20 alloc*, *structure access*, *structure set!*, *type access* and *prop-access*. The *structure alloc* program allocates a vector and sets every field to a new point structure with 2 fields. To do this, the baseline version allocates a vector and sets each field to a reference to the result of a call to the structure constructor. The version using structure vectors will allocate one (and initialize the type tree) and then set internal fields using *point-vector-set!*. The same operation is done on structures of 20 fields in *structure20 alloc*. We also measure the time spent on garbage collection (*structure20 alloc gc*) and on time taken without factoring the initial allocation of the vector and the initialization of the type tree in the new primitive's case (*structure20 alloc no-init*). The *structure access* program repeatedly accesses every structure stored in a plain vector of individually allocated structures or a structure vector, *structure set!* sets them to a new value instead. The *type access* and *prop-access* programs access the type descriptor or field of a single structure that is either an internal reference or a normal structure using the typical accessors defined by *define-type*. An obvious solution to large sets of small structures would be to ditch the structure mechanics and write theirs fields inline in a vector of size #fields $\times$ #structures. These fields are not compatible with the usual structure operations, but benchmarks are included where appropriate to show how the basic operations would compare with structures and structure vectors.

Three other programs, *convex envelope*, *quicksort* and *distance sort* are also used to cover more normal use cases. The first uses *Jarvis'* algorithm [2] to calculate the convex envelope of a set of points in a plane. The others uses selection and quick sort to sort a set of points by distance from the origin. We use a naive sorting algorithm to have somewhat of a worst case with regard to the ratio of accesses to allocations since this sorts a relatively small set of points. We also ran *quicksort* and *convex-envelope* using *(declare (safe))* instead to evaluate the performance hit caused by the runtime type checks.

4.2 Results

The results presented in Table 1 correspond to the implementation using normal vectors (*baseline*), the results for *structure vectors* and the ratio of *structure vector/baseline* (*ratio*). The implementation using normal vectors with each fields written directly are under the column *vector*. The subcolumn 32 and refers to 32 bits and 64 bits results respectively.

Allocating large numbers of structures by using structure vectors is quite fast. It takes roughly 2% of the time taken by the baseline version. The time taken to allocate larger structures shows improvement for the baseline because allocating fewer larger chunks puts less pressure on the garbage col-

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

(define count 4000000)

(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))))

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

Figure 10: Baseline *structure alloc*

lector. The contained version still takes only approximately 12% of the time the baseline takes. We notice that, in the baseline program, the majority of the time is spent in garbage collection whereas the structure vector version spends almost all of its time mutating the container along with a substantial time spent initializing the container tree in 64 bits. Using vectors containing the fields directly instead of structures yields similar performance to structure vector albeit slightly worse due to separate calls to *vector-set!* for every fields and the arithmetic required to compute the offset.

Accesses to contained structures and their fields takes between 1.23x (32 bits) and 1.48x (64 bits) as long as the baseline versions while mutating all the fields in contained structures takes less than a third of the time taken for normal structures. Accessing the type descriptor of internal structures is, as expected, much slower (3.59x and 4.51x). This requires traversing the container tree, thus doing several indirection versus a simple field reference made directly on the structure. For the normal vector alternative, we obviously cannot access the type descriptor since it is not stored, but access to fields is faster then both the baseline and structure-vector implementations.

The *convex envelope* benchmark is slightly faster using a structure vector as large amounts of allocations are performed and balance the actual computation which use many references. In safe mode, the cost of these references is larger because of the extra cost associated with type checks and make the version using a structure vector slightly slower then the baseline.

On the other hand, we have minimal gains (ratios of .87 and .85 for 32 bits and 64 bits) on the *distance sort* benchmark since we allocate only a few thousand *points*. The slight overhead on accesses probably compensates for most of the gains made in allocation. For the more efficient *quicksort*, with its much larger set of points, the ratios vary from close to one in unsafe mode to around two in safe mode.

The differences in performance between the baseline and structure vector versions follow roughly the same trends

	baseline		structure vector				vector			
	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**)				

Table 1: Benchmark results

in 32 bits but are somewhat more pronounced on both extremes. With the much reduced costs of initializing and maintaining the container tree, allocations observe over 60x speedup for small structures compared to the baseline benchmark and a 10x speedup for larger structures. Overall mutations, allocation, field access, type access and distance sort observe speedups from 64 bits while structure access and convex envelope are more expensive. Type accesses' overhead (ratios of 3.59 vs. 4.51) is slightly reduced by the shallower container tree despite the much smaller amount of work needed.

We also compared Gambit with structure vectors against the implementation without on the normal benchmark suit to measure the impact of the multiple structure type tags on generic use cases. We found that the average running time increased by 4% in 32 bits and 9% in 64 bits.

5. Related Works

The idea of compacting data representation by grouping similar objects together is not new. Region allocation[4–7] is somewhat commonly done in statically typed languages, whether manually or automatically. Other approaches toward reducing individual structures' size have been tried in Scheme like Chez Scheme's *ftypes* [3] that use structures of foreign data (like smaller integers for instance) similar to C structs. A similar system allowing statically typed fields in structures has also been implemented in Gambit Scheme and is orthogonal to structure vectors. The container tree algorithm is also largely based on the Multics multilevel paging system introduced in 1975 [1, 8] and frequently used in operating systems.

5.1 Multilevel Paging System

Paging systems split the whole memory in discrete chunks (pages) and use an indexing scheme to recover the appropriate page when a reference is made to its content. Multilevel tables separate the reference into groups of bits and use those groups to index the various levels in a tree of tables. This is also essentially what is done by our type tree algorithm with the reservation that we do not need (nor want) to index the whole memory and that our pages won't all be of the same size. This implies that we need a mechanism to dynamically add or remove subtrees when necessary in order to preserve memory.

5.2 Allocation by Regions

Allocating objects of the same type together in memory is a common strategy to facilitate memory reuse and data locality. Several approaches are used ranging from manually managed object pools [6] to statically managed regions performing automatic memory management through a variant of typed lambda calculus [5]. Those approaches are used in statically typed languages and make use of type erasure for efficient representation. They don't need to bother with run time type checking so objects don't have to include typing information and allocation by regions doesn't alter the representation of objects.

5.3 Chez Scheme's Ftypes

Keep and Dybvig have introduced C struct analogs in Chez Scheme to allow interoperability with C functions. These are used as part of the *FFI* to specify data structures with statically typed fields. These fields permit more compact representation of structures by possibly using only the required number of bits and discarding tagging. These structures still require a header and typing information to allow garbage collection and must be allocated individually.

6. Conclusion

The implementation of structure vectors in Gambit Scheme provides programmers with a way to significantly reduce the memory footprint of large sets of small structures and group them for better locality. This is done by allocating the structures in one go in a vector and adding the header and type descriptor only to the vector instead of keeping this information on every single instance of the structure. This more compact representation allow the allocation of n structures of size f to take up only $2 + fn$ words instead of the $n(2 + f)$ words taken by the normal allocation method.

We introduce a multilevel container tree indexed using the bits in references to unconditionally recover the type descriptors in constant time. This container tree is kept updated on the allocation and freeing of structure vectors by hooks added to the *runtime* and is exposed to the Scheme program through primitives that are used during dynamic type checks and accesses. This allows the contained structures to be used transparently with the existing structure primitives dealing with the type descriptor.

To make the structure vectors available to the programmer, we have introduced a set of new primitives and modified the *define-type* macro to generate functions or macros that use them. These primitives include a constructor and getter and setter for contained structures. We also ensured that every method dealing with structures could take an internal reference without modifications.

Our performance evaluation demonstrate that gains can be had using structure vectors. On a 64 bits platform, we observe significant speedups in allocation time and structure mutation using structure vectors and limited overhead for most other operations. Predictably, there is a slowdown (4.51x) on accesses to the type descriptor of internal structures and to contained structures (1.48x). In 32 bits, the trend is similar but more pronounced, with large speedups on allocation and mutations and minor gains to slight overhead for most other operations except type accesses with a 3.59x slowdown.

Acknowledgments

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada.

References

[1] A. Silverschatz, and B. Galvin. Operating System Concepts, 5th Edition, Wiley, 1999

[2] R. A. Jarvis. On the identification of the convex hull of a finite set of points in the plane, Information Processing Letters, vol. 2, no. 1, p. 18-21, 1973

[3] R. K. Dybvig, and A. W. Keep. Ftypes: Structured foreign types, Workshop on Scheme and Functional Programming, 2011

[4] R. Jones, A. Hosking, and E. Moss .The Garbage Collection Handbook: The Art of Automatic Memory Management, Chapman & Hall/CRC, 2011

[5] M. Totfe, and J.-P. Talpin. Region-based Memory Management, Information and Computation, vol. 132, p. 109-176, 1997

[6] D. Gay, and A. Aiken. Memory Management with Explicit Regions, SIGPLAN, vol. 33, p. 313-323, 1998

[7] C. Lattner, and V. Adve. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap, SIGPLAN, vol. 40, p. 129-142, 2005

[8] B. S. Greenberg, and S. H. Webber. The multics multilevel paging hierarchy, Multics Technical Bulletin, vol. 170, 1975