Record Data Structures in Racket
Usage Analysis and Optimization

Tobias Pape
Hasso Plattner Institute
Universtiy of Potsdam
tobias.pape@hpi.uni-potsdam.de

Vasily Kirilichev
Hasso Plattner Institute
Universtiy of Potsdam
vasilii.kirilichev@hpi-alumni.de

Carl Friedrich Bolz
Software Development Team
King's College London
cfbolz@gmx.de

Robert Hirschfeld
Hasso Plattner Institute
Universtiy of Potsdam
hirschfeld@hpi.uni-potsdam.de

ABSTRACT

Built-in data structures are a key contributor to the performance of dynamic languages. *Record data structures*, or *records*, are one of the common advanced, but not easily optimizable built-in data structures supported by those languages. Records may be used in an object-oriented fashion or to implement object orientation itself.

In this paper, we analyze how records are used in different applications in the Scheme dialect Racket. Based on the data obtained, we suggest the application of existing optimization techniques for records and devise a new one for immutable boolean fields. Most of them can be applied to a wide range of record implementations in dynamic languages. We apply these optimizations to records in Pycket, an implementation of Racket. With one exception, micro-benchmarks show a two- to ten-fold speed-up of our implementation over plain Racket.

CCS Concepts

•Information systems → Record and block layout; •Software and its engineering → Data types and structures; Classes and objects; *Just-in-time compilers;*

Keywords

Record data structures; Objects; Racket; Optimization

1. INTRODUCTION

For programming language implementations, performance is often key and, among other aspects, built-in data structures contribute to the overall performance of a language implementation. The lack of optimization of built-in data structures may result in poor performance and increased memory consumption of dynamic languages [2, 18]. In the context of modern virtual machine (VM) development frameworks, such as RPython, some data structures, such as collections [5], are already in the focus of research.

http://dx.doi.org/10.1145/2851613.2851732

Record data structures or *records* are one of the advanced common built-in data structures, which are not deeply investigated in the sense of optimizations for modern VMs. Basically, records aggregate heterogeneously typed, named fields, possibly with a definition in a record type. In some languages, such as Racket, records may not only be used to store the data, but have additional features. Racket is a dynamic multi-paradigm Scheme-family programming language with powerful built-in record data structures, where records can behave like objects of a class or even like a function. Records also often provide identity, encapsulation, abstraction, and maybe behavior, thus providing key ingredients for object orientation. In fact, records can be used to implement object-oriented features, such as the class-based object orientation in Racket [13].

A simple, straight-forward implementation of records for dynamically typed languages implies a big overhead because of the semantics complexity. It is more important for the implementation to be simple than the interface. Thus, many languages prefer the "worse-is-better" approach [14], whereby the simplicity and efficiency of implementation are more important than the straightforward following the semantics and perfect correctness. Our analysis shows that, at least for the Racket language, records have a noticeable optimization potential. In this work, we consider an efficient implementation of records for dynamic languages and for Racket in particular. We focus on the RPython-based implementation named Pycket.

In this work, we make the following contributions:

- We analyse and evaluate the usage of record data structures in Racket applications (section 3).
- We identify applicable optimization techniques for the efficient implementation of record data structures (section 4). In particular, *we propose a novel optimization technique for static immutable boolean fields in record data structures* (section 4.3).
- We implement Racket's record data structures with optimizations and evaluate performance results (section 5 and 6).

2. BACKGROUND

Record data structures, or *records*, are collections of named fields of heterogeneous values. Records may form a type, instances of record

Listing 1: Racket structures using structure hierarchies, explicit mutability, callable structures, and prefabs.

```
1  (struct person (name))
2  (define customer (person "Sam Adams"))
3  (struct employee person (position [salary #:mutable])
4    #:property prop:procedure
5    (lambda (self) (* (employee-salary self) 0.146)))
6
7  (define worker (employee "John Smith" "Developer" 50000))
8  (person? 0) ; -> #f
9  (person? customer) ; -> #t
10 (person? worker) ; -> #t
11 (employee? customer) ; -> #f
12 (employee? worker) ; -> #t
13
14 (set-employee-salary! worker 55000)
15 (employee-salary worker) ; -> 55000
16
17 (worker) ; -> 7300
18
19 (define john-station
20   '#s(workstation "station01" "fd23:5e15:aa18::2" 5))
21 (struct workstation (name ip age) #:prefab)
22 (workstation? john-station) ; -> #t
```

types are typically of equal size — all in contrast to data structures like arrays that are collections of typically indexed fields of homogenous values. Array-like data structures do not form types. Individual arrays may differ in size. Moreover records may have various additional features, which may differ between programming languages.

2.1 Structures in Racket

Racket [12] is a dynamically typed, multi-paradigm programming language from the Scheme-family [21]. Racket differs from Scheme in certain aspects such as immutable-by-default lists, built-in support for *design by contract* [17], or a more complex record data structure concept called *structures* (or *structs*), providing features beyond the mere ability to store values in their fields.

Racket structure types can form *hierarchies*, supporting inheritance. Structures in Racket are *immutable* by default, but can be explicitly declared to be partly or fully mutable. Structure type *properties* allow to store arbitrary data inside the structure type. However, typically properties are used for procedures that work on a structure's field values. Certain properties can be used to make structure instances *callable*; these structures can then act like procedures. Other properties denote structures as *transparent*, allowing run-time reflection on a structure's internals. Racket also supports a shortcut form of structures for literal specification of structures before the formal introduction of their structure type. These structure types are called previously fabricated structure types (*prefabs*).

The example in listing 1 contains two structure instances: a `person` named "Sam Adams", bound to `customer` in line 2. The corresponding structure type `person` is defined in line 1 as structure with one field, `name`. The predicate `person?` further down confirms this. The second structure instance bound to `worker` in line 7 is an `employee` named "John Smith" (`name`) in the "Developer" position (`position`) who earns 50 000 money (`salary`). The structure type `employee`, defined in line 3, makes use of structure hierarchies — it is a sub-type of `person` and inherits its `name` field. Moreover, it has a *mutable* field `salary`. Hence, the mutator `set-employee-salary!` further down can be used to update the field. The accessor `employee-salary` can be used to retrieve the stored value. Then, the structure type has a *property* named `prop:procedure` that is bound to a procedure. That way, *calling* the `worker` structure instance in the last line results in the procedure to be called with this instance and computes the amount of medical insurance fee based on the salary and the fixed rate. Lastly, `john-station` is defined as a *prefab* structure in line 20, without the need (but possibility) to define the structure type `workstation` beforehand. However, such a structure type can be defined after the fact, as in the line following. The predicate `workstation?` confirms that, indeed, `john-station` is of the expected structure type.

2.2 Structures and Objects

Scheme is a multi-paradigm language family that is probably best known for its functional aspects. However, object-orientation is not only possible to implement and use, for example with Common Lisp Object System (CLOS) implementations such as TinyCLOS, in Racket an object-oriented (OO) implementation is readily available with the `racket/class` standard library. It provides class-based object orientation with message passing, mixins, and traits [13]. This system is implemented in terms of Racket structures; every class is also a structure type, every object is a structure instance. While it would have been possible to focus solely on the object-oriented part of Racket, considering all structures instead benefits the implementation of object orientation as well as other parts of Racket.

Racket structures actually can directly be used in an object-oriented fashion — at the loss of message passing and run-time polymorphism compared with the library implementation of object orientation. However, other object-oriented fundamentals, such as instance identity, encapsulation, abstraction, and even object behavior are already present in Racket's base structures and also justify an investigation under an object-oriented point of view.

3. STRUCTURE USAGE IN RACKET

Racket structures are a powerful data structure with broad applicability. They are widely used in Racket packages[1] and projects on GitHub[2]. Structures are essential for the Racket contracts implementation. In this section, we investigate how structures are actually used in different Racket applications. We perform a static and dynamic analysis of existing applications to identify the typical size of structures, types used within structures and the frequency of mutation.

We choose five Racket applications from different domains including development tools, text analysis, mathematics, and games. *I Write Like*[3] — one of the biggest Racket applications — is a web appication that analyses the style of a given text by comparing with styles of many famous writers. This application represents a heavy text analysis application. The *markdown* parser application[4] is a simple parser for *markdown* formatted text that is used in many other Racket projects as a library. *Racket CAS*[5] is a simple computer algebra system for Racket with a good built-in test set. *2048*[6] is a Racket implementation of a famous puzzle-game with numbers. Finally, *DrRacket* is a feature-rich Racket integrated development environment (IDE), which is widely used by Racket-programmers.

[1] http://pkgs.racket-lang.org (visited 2015-12-05)
[2] https://github.com/search?q=language%3Aracket (v. 2015-12-05)
[3] https://github.com/coding-robots/iwl (visited 2015-12-05)
[4] https://github.com/greghendershott/markdown (visited 2015-12-05)
[5] https://github.com/soegaard/racket-cas (visited 2015-12-05)
[6] https://github.com/danprager/racket-2048 (visited 2015-12-05)

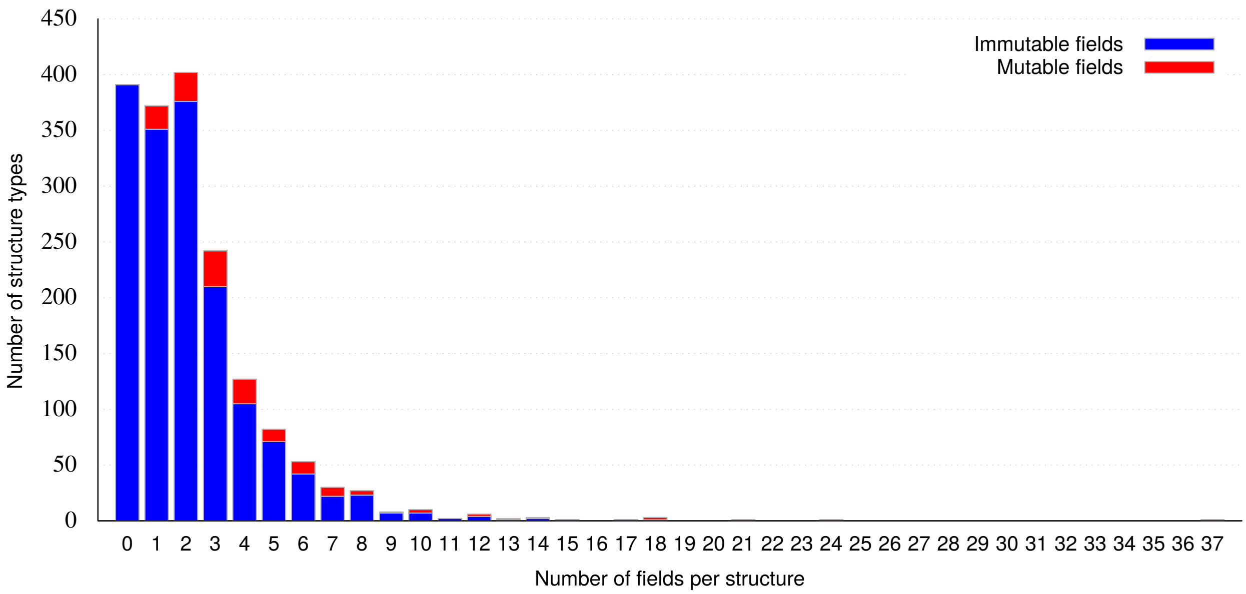

Figure 1: Distribution of number of structure fields in the Racket standard library.

Table 1: Results of the static analysis of Racket standard library source code files.

Structure Type	1765	100 %
With super-types	563	31.9 %
With mutable fields	148	8.4 %
Transparent	659	37.3 %
Prefabs	146	8.2 %

3.1 Static Analysis

We perform a static source code analysis of the Racket v6.2.0.4 standard library comprising 4 812 Racket source code files. We track the number of immutable and mutable fields and super types per structure.

3.1.1 Results

Of all the source files, 11.6 % contain all 1765 structure type definitions (cf. Table 1), 31.9 % with super-types. Structures have 2.3 ± 2.6 fields on average, with a median of 2. The largest structure from the Racket library has 37 fields. 91.6 % of all structure types are immutable. Structures with mutable fields tend to be larger (maximum: 37, mean: 4.55 ± 4.56) than all-immutable structures (maximum: 18, mean: 2.10 ± 2.17). The distribution is shown in Figure 1.

The statically determined number of structure types in the applications analyzed is comparatively small; together, they define 22 structure types with at most 5 fields (average 1.64 ± 1.26, median 1), all immutable. We refrain from plotting the distribution.

3.2 Dynamic Analysis

We instrumented the structure implementation in Racket to track the creation process of structure types, structure instances, the amount and types of structure field values, and the frequency of mutate operations. Our analysis reports the total usage of structures including the Racket core.

3.2.1 Results

Refining the static analysis, about 85 % of all fields used are immutable, with *DrRacket* being an outlier with about 61 % of immutable fields. Structure instances have 1.62 fields on average with a median of 1. The number of instances of each structure type depends heavily on the specific application, ranging from 200 to 1500 in our tests. The number of mutations varies even more.

Although structures in Racket are typically used monomorphic, that is the data type of values stored in a field does not change, some instances' fields are used with values of more than one data type (*non-monomorphic*). The amount of structures containing at least one non-monomorphic field is between 5 % and 15 %.

The distribution of field types is homogeneous as illustrated in Figure 2. The most common data type used in structure field type is *boolean*. Up to 70 % of booleans have the value `#f` (false), which is used in up to 88 % as a placeholder default value for other data types, such as *procedure*. *Procedure*s are also used widely, to the extent that some structures only contain exactly one procedure—such procedure-containers are often used as super-types for other structures. *String*s, mutable and immutable, pose the most user-faced data type in field types while *symbol*s and the *syntax* type (used by the Racket macro system) are more system-faced, or even meta-level types used in structures. Non-scalar field types, such as *pair*s and *list*s, and other *structure*s are common as field types, too. Other types have a collective share of about 10 %.

Despite our initial assumption, *integer*s are not very common, except for the *2048* game that heavily uses *integers* and *floats*. Other applications use numbers significantly less frequently. To show this, we separated *2048* in Figure 2.

We found only few common data type collocation patterns in structures, despite the homogeneous field type distribution. Such patterns

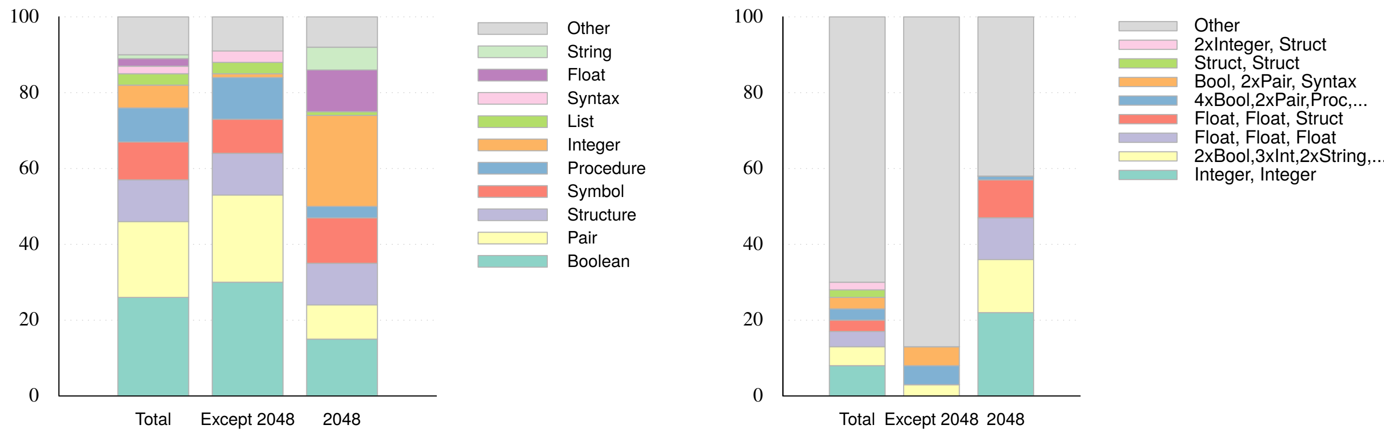

Figure 2: Most frequent field types (left) and most frequent combinations of field types (right) in Racket applications

include the use of *integer, integer*-structures in *2048* for coordinates, the most prevalent collocation in this application. This is, nevertheless, uncommon for other applications. Thus, combinations of stored together field types in structures are mostly application specific. No patterns can be derived in the general case as less than 30 % of all structures exhibit significant similarity. The right part of Figure 2 shows this in more detail.

3.3 Discussion of Analysis Results

We found that Racket structures are relatively small and contain between one or two fields on average. Furthermore, about 85 % of structure fields are immutable. Initially unexpected, *booleans* are the most common data type in structures. We found that `#f` (Racket's *false* value) is used a placeholder default value and that the corresponding filled value is often a procedure.

4. OPTIMIZING RECORDS

Based on the analysis in section 3, we propose fitting optimizations to use to improve performance when compared with a simple, straightforward direct-mapping approach. We think that this catalogue of optimizations can be worthwhile beyond Racket, given the usage of record data structures is not completely dissimilar. In particular, we suggest applying four standard optimizations and propose a new one, immutable boolean field elision (IBFE). As a running example, we will use the structures of listing 1.

4.1 Direct Mapping Approach

We first present a most simple approach to realizing Racket structures, by directly mapping the semantical language components to memory entities. Considering Figure 3, directly applying Racket's semantics, we end up with two records instances, one for the `employee` type and one for the `person` type. The field values are stored in the storage objects of both instances according to its type. Using storage arrays poses a simplification as records with different numbers of fields can be represented by the same implementation type. Note that this approach does *not* constitute best practise but rather serves as a baseline for the optimizations to come.

This approach anticipates the Racket way to access inherited fields. For example, to access the `name` of `worker` in our example, the native

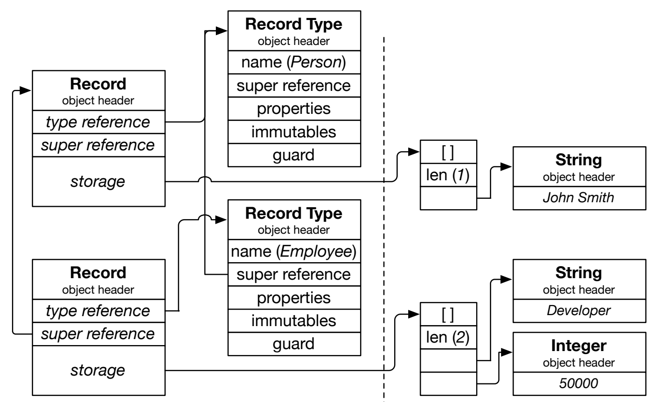

Figure 3: Direct mapping approach representation of `worker`

accessor behavior will be called with an offset 0 and the structure type `person`, but to access `position`, it will be called with an offset 0, *too*, but this time with the structure type `worker`. However, certain performance improvements already become apparent: super-instances never exist solitarily but always together with their sub-instances. Furthermore, they duplicate the hierarchy information already available in the type.

4.2 General Optimizations

We first consider and apply existing optimizations for records and similar structures. The combined approach is illustrated in Figure 4.

4.2.1 Flat Structure

A flat structure collapses the semantical hierarchy of record objects and represents every record with only one object that combines all fields in its storage. Such an implementation is typical for objects in OO languages, for example Squeak / Smalltalk [15]. This approach loses the redundant super-instance / sub-instance tandem and hence improves memory consumption.

A flat structure saves memory by removing redundant record objects. A simple, straight-forward record object has a size of four machine-words multiplied by the number of super-types, one for the header

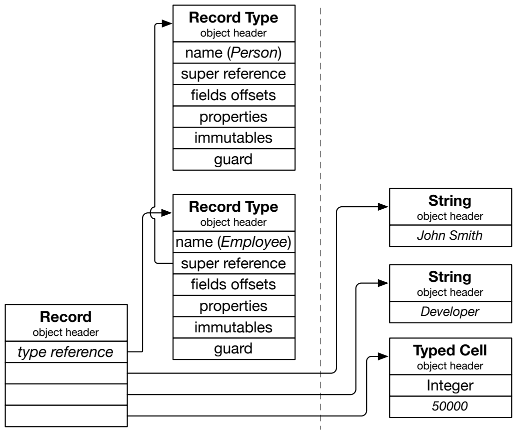

Figure 4: Flat structure, inlined fields, and mutable `salary` field, wrapped into a *typed cell* with an unboxed value.

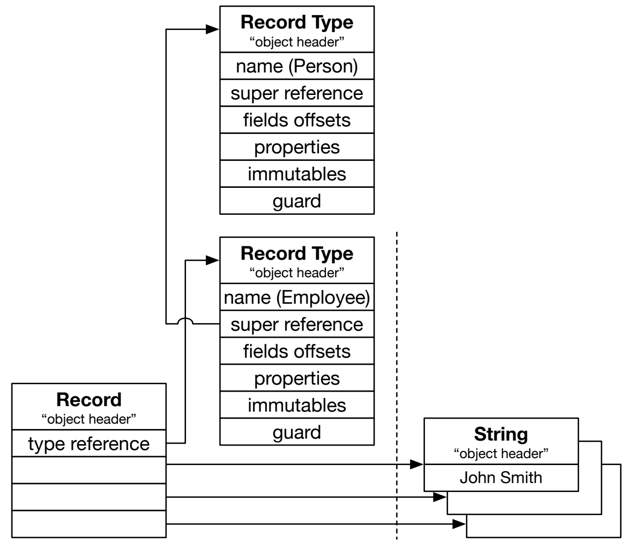

Figure 5: Employee record data structure with inlined fields.

and three for references to the type, super instance, and storage for each record object in the hierarchy. Additionally, every record object typically has a storage that takes two additional words (typically one for the length and one with special purpose, commonly for the garbage collector (GC)). Records with a flat structure do not have super-instances, which saves one word for the reference to the super-type and six words for every super-type of every record instance, that is four word for the record and two for the storage. Assuming that about 31.9 % of Racket's structures have one super-type (cf. section 3), for *n* structure objects, that saves

$$n \cdot (6 \cdot 0.319 + 1) \quad (1)$$

words in Racket on average.

Nevertheless, records with a flat structure make the implementation of the native accessing behavior more complex. The per-structure-type indices now have to be mapped to the absolute index into the record's storage. These indices do not change over time and, therefore, a static mapping for each field can be calculated in advance.

4.2.2 Inlining

The direct mapping approach contains an indirection between a structure's representation entity and the actual storage for the structure's fields. This eases the implementation of the representation entity, for example as instances of a structure class. This additional hop, however, can be cause for performance bottlenecks, as every field read has to traverse the indirection. A best practice is to fuse records and their storage, improving execution time performance by reducing costs of object allocation and pointer dereference. Implementations like the Squeak VM or the Java Virtual Machine (JVM) do this for their object representation.

Arbitrarily large structures may, however, slow down the overall allocation performance and hamper GCs. While Racket structures may have up to 32 768 fields[7] , the actual amount of structure fields used in Racket is typically low; between one and two fields on average (cf. section 3). Hence, we propose to limit inlining to only few fields and store larger records with a separate storage, as done in the PyPy implementation of Python [20].

[7] http://docs.racket-lang.org/reference/creatingmorestructs.html (visited 2015-07-01)

Field inlining reduces the complexity of data access operations by removing a one additional access hop. This optimization releases three words for every structure instance, that is one word for a storage reference and two words for a storage array (length and special/GC word, as above). For *n* structures, that saves

$$n \cdot 3 - sizeOfSpecializedClasses \quad (2)$$

words, where the *sizeOfSpecializedClasses* indicates the memory needed for specialized record classes containing a certain amount of fields. If the total amount of fields is greater than a predefined limit, fields are stored in the storage without changes. This avoids the creation of a big amount of record classes, which may not be used at all. The inlining technique has two important advantages. First, records with inlined fields take less space than records with a storage. But even more, inlining is crucial for optimizing the access to the record fields, because it avoids an indirection to a separate storage.

4.2.3 Unboxing and mutability separation with cells

Record data structures in dynamic programming languages can contain fields of arbitrary types. A usual way to implement storing of different data types together is *boxing*, that is, an allocation of all field values on the heap with a common header. Boxing simplifies the implementation of dynamic programming language significantly, because all different objects obtain a common simple representation. Nevertheless, boxing is not always efficient.

For example, to store an integer in C, only one machine-word is typically needed. For comparison, it typically needs at least three words to store a boxed integer in a dynamical language running in the VM (one to store the type of object and the last one to store the particular value of integer, and typically one for the GC) [5]. This problem gains importance when many objects are stored together in records.

One solution is to store the record field values *unboxed*, saving the type information separately once for many objects. The object where this type information is stored is called *field type*. However, a mutation of a field to a new value with another type involves dereferencing of

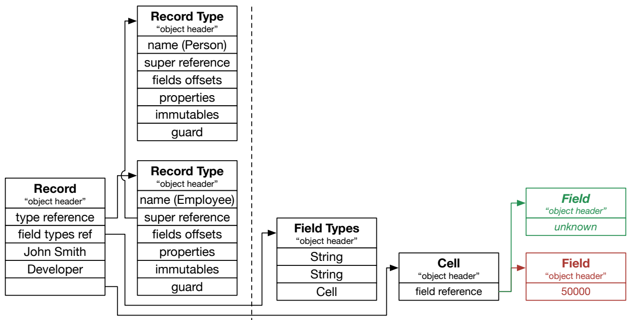

Figure 6: Record with a mutable `salary` field, wrapped into a *cell*. Mutation does not affect the immutable reference in the structure — rather, the reference from the cell to the old value (red) will be updated with a new value (green italic).

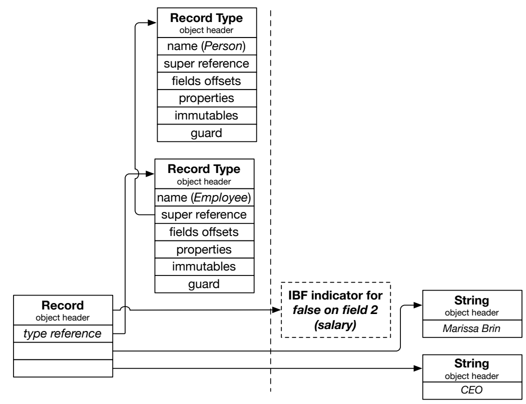

Figure 7: `employee` structure with an IBF indicator denoting the elision of field 2

a *field types* object and even a creation of a new *field types* object if no proper *field types* object exists.

Assuming that structures have 1.6 fields on average and the part of structures with homogenised fields is about 85 %, for n structures, this optimization saves

$$n \cdot 2 \cdot 1.6 \cdot 0.85 \cdot 0.44 - n - sizeOfFieldTypes \approx 0.2n \qquad (3)$$

words in Racket, where $sizeOfFieldTypes$ is the size of *field types* objects containing the type information of fields. That is, this optimization is not very efficient for small records in Racket. Furthermore, unboxing brings some memory overhead because of storing *field types* pointers and increases the implementation complexity. This overhead may lead to the negative optimization effect in the worst case, when records contain boxed values only.

The complexity of *field types* to support mutability can be alleviated by partially boxing the mutable content of structures. Also, implementations can take advantage of the fact that Racket structures have mostly immutable fields. Moreover, if *all* fields of a structure were always immutable, better optimizations would be imaginable; especially just-in-time (JIT) compilers that use tracing or partial evaluation could benefit. Combining this, we propose to treat all structures as immutable and use an indirection object, called *cell*, for the few fields that are actually mutable. Changing the value of a field no longer affects the structure itself but rather delegates the change to the cell representing the mutable field, as can be seen in Figure 6. That way, the maintenance of *field types* is completely absorbed into managing cells, which may or may not be typed. This technique is common in Lisp and Scheme applications, among others. As the mutability of fields is a property of a structure's *type*, wrapping objects in cells can efficiently be done at structure allocation.

Using cells implies an inherent memory and access time overhead. However, as most fields are used monomorphic, we can specialize cells to *typed cells*, which store a type and an unboxed value. They can change their type field dynamically upon mutation. Thus, if a mutable record field belongs to a known type, such as integer or float, a typed cell stores its value unboxed, reducing the cell's overhead.

We obtain an optimization that (a) improves execution time for access to mutable fields and (b) can reduce memory consumption for larger structures. However, the general structure size is small in Racket, so we propose to just use the *cell* part without the general unboxing.

4.3 Immutable Boolean Field Elision

Booleans are the most frequent field type in Racket structures. However, up to 70 % of boolean fields have the value `#f`. Knowing that most (up to 85 %) fields are actually immutable, a high number of fields in Racket structures hence consist of immutable boolean fields (IBFs).

It seems feasible to actually *not* store this infomation as a field value per se. Instead of storing both positions and values of the boolean fields, we use an indicator to denote all positions of IBFs within a structure, effectively *eliding* the immutable `#f` values; we call this immutable boolean field elision (IBFE). This indicator might be implementation specific; but in the same way structures that contain mutable fields or unboxed fields must be commuticated to the runtime, IBFs can be communicated similarly, be it tagging, header bits, or class-based indication as in Figure 7, to name a few. It is crucial that all possible combinations of IBFs for an arbitrary record instance are present as indicators at structure allocation time. For example a record class with three fields, all immutable, that gets instantiated with an `#f` value on position two could use an implementation class that treats position two specially by not providing storage for it (cf. Figure 7). That implementation class would act as IBF indicator. Note that the `#t` value is not treated specially by immutable boolean field elision (IBFE), as are `#f` values in mutable fields. These are stored as if IBFE was not present at all.

The booleans optimization saves memory by reusing immutable *false* values. Assuming that structures have an average size of 2.3 fields, 26 % of all fields are booleans and 70 % of booleans are *false*, and also that 85 % of fields are immutable in Racket, for n structure objects, this saves

$$n \cdot 2.3 \cdot 0.26 \cdot 0.7 \cdot 0.85 - sizeOfSpecializedClasses \approx 0.36n \qquad (4)$$

words in Racket on average, where $sizeOfSpecializedClasses$ indicates the required memory for pre-defined structure classes with *false* fields. Although this optimization may have less positive impact on memory consumption on average, it does not add memory overhead for records in the worst case as unboxing with *field types* would. For extreme case, where every record has one immutable field with a value *false*, the saving would be approximately n.

Using IBFE, memory for immutable `#f` values can be saved at the expense of providing a large enough number of IBF indicators, which poses a trade-off. Applications with only few IBFs and large structures would be hit by the overhead of maintaining IBF indicators; however, our analysis shows that these cases are rare in Racket applications.

5. STRUCTURES IN PYCKET

We implemented the presented optimizations in Pycket, a Racket implementation using the RPython toolchain and its meta-tracing JIT compiler.

5.1 RPython and Pycket

RPython [4] is a framework for implementing interpreters, consisting of a type-inferenceable ("restricted") subset of Python and a toolchain that *translates* an interpreter written in the RPython language into an efficient VM. Lower-level VM features, such as GC, object layout, and a meta-tracing JIT compiler are inserted automatically during the translation process. RPython was used for efficient implementations of several dynamic languages including Python [1], Prolog [7], and Smalltalk [6].

5.1.1 Meta-tracing

Typically, tracing JIT compilers optimize the executable program directly. For the meta-tracing JIT, the executable program is itself an interpreter running a user's program code. In other words, the meta-tracing JIT operates on a representation of the interpreter. In order to produce efficient VM with RPython, the interpreter needs some hints from the developer, to help the tracing JIT to identify loops in the interpreted program, and perform other optimizations.

The RPython JIT compiler records operations executed by the interpreter running a user's program. The produced linear sequences of machine code are called traces and they are only recorded for loops, which were performed more than a certain number of times. Only operations executed by the interpreter for one iteration of the loop are recorded by the tracing JIT compiler. The tracing process then optimizes the machine code instruction sequence and generates new machine code that is used for next iterations of the loop.

Because of the linearity of sequences, the trace represents only one of the potential execution flows. To provide the correctness of program execution, the JIT inserts *guards*, special instructions, which detect when the program execution conflicts with the trace and return control back to the interpreter.

5.1.2 Pycket

Pycket[8] is an implementation of Racket using the RPython toolchain and based on the control, environment, and continuation (CEK) abstract machine [11]. Using the CEK machine eases the implementation of some more complex features of Racket, such as proper tail calls, first-class continuations, and multiple return values [3]. It is already competitive with the best existing ahead-of-time (AOT) Scheme compilers, particularly on safe, high-level, generic code [8]. However, it is not yet feature-complete and in particular had no structure support prior to this work.

5.2 Optimization Steps

Practically all implementations of record-like data structures skip the step Direct Mapping Approach described in section 4.1. However, for evaluation purposes, we included a direct-mapping-based implementation all following optimizations are applied to. Accordingly, all structure types are implemented as instances of an RPython-level class (`W_StructType`) and all structures as instances of a distinct class (`W_Struct` or its subclasses) with a reference to the structure type, a references to a storage for the fields, and possibly a reference to its super-instance.

5.2.1 Flat Structure

For a flat structure, a structure instance no longer refers to its super-instances but assumes all their former fields. However, the positions of all fields in the structure type hierarchy have to be mapped to the absolute fields positions to retain data access semantics: the language-level accessor and mutator procedures handle field indices relative to their respective type and respective to the top-most super-type. The absolute offsets do not change, however, and hence are calculated once during the structure type initialization and marked as immutable. This allows the JIT compiler to remove most field-position related calcualtions at run-time.

5.2.2 Inlining

The inlining optimization changes the data layout of structures by fusing structures and their storages together. This optimization requires the creation of modified structures, which may contain field values as attributes.

To inline fields into the structure instance, several specialized structure classes exist that each represent structures of a certain size. Following PyPy's example, only up to 10 fields are actually inlined; larger structure instances still use a separate storage. Therefore, 12 implementation classes for structures are provided. The decisions which particular class is used for a structure instance is made at run-time as part of the instantiation process. Thus, if a new structure does not exceed the limit, one of the specialized implementation classes is chosen, and field values are saved in the structure's attributes directly.

5.2.3 Typed Cells for Mutability Separation

Cells allow to keep all structure fields immutable by wrapping all mutable fields into cell objects. Cells stay immutable itself as a part of the structure, but may change their content.

The concept of a typed cell was already available in Pycket before introducing structure support and has been used for mutable globals and environment optimization, to name a few. Pycket cells store their values unboxed using *storage strategies* [5]. If a matching strategy exists, a cell stores its value unboxed, for example integer and float values. Otherwise, cells use a *general strategy* and store values boxed.

Hence, for structure support, upon creation of a structure instance, all mutable fields — which are known in advance — are wrapped by cells and all of the structure instance's actual fields stay immutable. Also, all accessor and mutator behavior has been adapted to use the cells to unwrap and wrap valued automatically.

5.3 Eliding Immutable Boolean Fields

To benefit from immutable boolean fields, we suggested immutable boolean field elision (IBFE) in section 4.3. We chose to use the structure implementation class to represent the IBF indicators. As RPython does not support creation of RPython-level classes at run-time, all necessary indicators have to be generated in advance, before translation. However, a very high number of IBF classes can severely slow down allocation and possibly start-up time. Therefore, we assume an upper limit to the number of fields we consider for IBFE.

[8]https://github.com/pycket/pycket/ (visited 2015-06-01)

The amount of indicators that are necessary for a given limit l is $\binom{l}{1} + \binom{l}{2} + \cdots + \binom{l}{l}$. In Pycket, we chose 5 as the default limit, resulting in 21 pre-defined IBF indicator classes. This seems sufficient, given the average size of Racket structures but not overly restrictive, as it covers over 90 % of the structure type encountered in the Racket standard library (cf. section 3). Nevertheless, all IBF indicator classes are subjected to the inlining described above, so that each IBF indicator is actually represented by 12 classes for the field inlining.

Hence, when instantiating a structure, Pycket has 252 structure classes to chose from. The operation that maps from all IBF positions to the matching structure class benefits from a lexicographical order of all structure classes; the combination of `#f` positions determines the position of a structure class uniquely. During instantiation, all positions of immutable fields about to be initialized with `#f` are shifted to account for their elision. This can also help the inlining optimization, as larger structures with many IBFs now can potentially use an inlined representation instead of a split one.

Accessing an IBF is cheap; with IBFE we make sure that all accesses to those fields are in constant time.

5.4 A Note on Unboxing

As outlined in subsubsection 4.2.3, providing unboxing and mutability with *field types* is expected to only help for larger structures. It turns out that Pycket already has provides limited unboxing capabilities for implementation classes *iff*

1. all fields are immutable,
2. the size is not larger than two, and
3. the stored values are either of Racket's *fixnum* or *flonum* type.

In this case, two or four words of memory can be saved. Given Racket structures are mostly small and mostly immutable. However, we use this automatic unboxing *only* for the IBFE optimization level.

5.5 Implementation Summary

Overall, the whole structures implementation in Pycket includes 15 implementation classes, about 30 structure primitives, and about 50 general primitives, totalling in about 2000 lines of RPython code.

6. EVALUATION

Pycket is not yet a feature-complete Racket implementation and due to pending (non-structure related) features, the existing Racket structure benchmarks do not run yet. We therefore use a set of micro-benchmarks[9] instead. We provide an evaluation and execution time and memory consumption based on these benchmarks.

Setup All benchmarks were run on an Intel Core i5 (Haswell) at 1.3 GHz with 3 MB cache and 8 GB of RAM under OS X 10.10.2. All micro-benchmarks are single-threaded. RPython at revision a10c97822d2a was used for translating Pycket. Racket v6.2.0.4 and and Pycket at revision 3d0229f were used for benchmarking.

Methodology Every micro-benchmark was run five times uninterrupted. The execution time was measured *in-system* and, hence, it does not include start-up time. However, it does include warm-up time and the time needed for JIT compilation. We show the execution times of all runs relative to Racket with bootstrapped [10] confidence intervals for a 95 % confidence level. The memory consumption was measured as maximum resident set size and is given relative to Racket; the confidence intervals were negligibly small and have been omitted.

6.1 Micro-benchmarks

The micro-benchmark set consists of of ten tests. Besides examining basic operations, such as structure creation, call of the predicate procedure and accessing and mutating structure fields, we include two slightly more realistic use-cases.

6.1.1 Basic Operations

We used the following benchmarks for the basic operations: *create* creates simple structures representing two-dimensional coordinates with integer values; *create/super* re-uses the *create* benchmarks, but adds a third dimension using structure type inheritance; *create** is the same a *create*, but with an IBF as first field; *create/super** is the same as *create/super*, but with an IBF as first field; *predicate* checks the type of given structures including the whole type hierarchy; *access* performs accesses to various immutable fields of structures; and *mutate* changes every value of a structure and reads the stored value afterwards on each loop iteration. Each benchmark essentially contains a loop with few basic operations and collects the result in a variable to avoid elimination.

6.1.2 Binary Tree

In the *binary tree* benchmark, the base structure type represents a leaf, which has only a value. A node is a subtype of the leaf referencing two other nodes. This benchmark tests several operation with structures of multiple types simultaneously. We use two versions of this micro-benchmark, where values of leaves are integers (*binarytree*) and booleans (*binarytree**), respectively.

6.1.3 Parser

The *parser* benchmark is a Brainfuck[10] interpreter. It creates one instance of a structure referencing a list and a data pointer. The operations on the structure include mutations of the data pointer and accessing list elements, and hence, the *parser* benchmark tests the structure's accessor and mutator, but not the constructor. The benchmark's interpreter executes a simple program that generates a Sierpinsky triangle several times.

6.2 Optimization Impact and Results

We report the impact of all optimizations on execution time and memory consumption. The final performance results of optimized Pycket are shown in Figure 8. Note that we accumulate optimization, as they form dependencies. Hence, for example, *inlining* includes *flat structures*. The raw numbers are presented in section 9. By way of example, we show the validity of the predicted memory saving, using the *create*, *create/super*, and *binarytree* benchmarks. For IBFE, we however use their boolean counterparts *create**, *create/super**, and *binarytree**.

6.2.1 Direct Mapping Approach

[9] https://github.com/vkirilichev/pycket-structs-benchmarks (visited 2015-12-05)

[10] Brainfuck is an esoteric programming language that models a Turing machine with eight operations on an array.

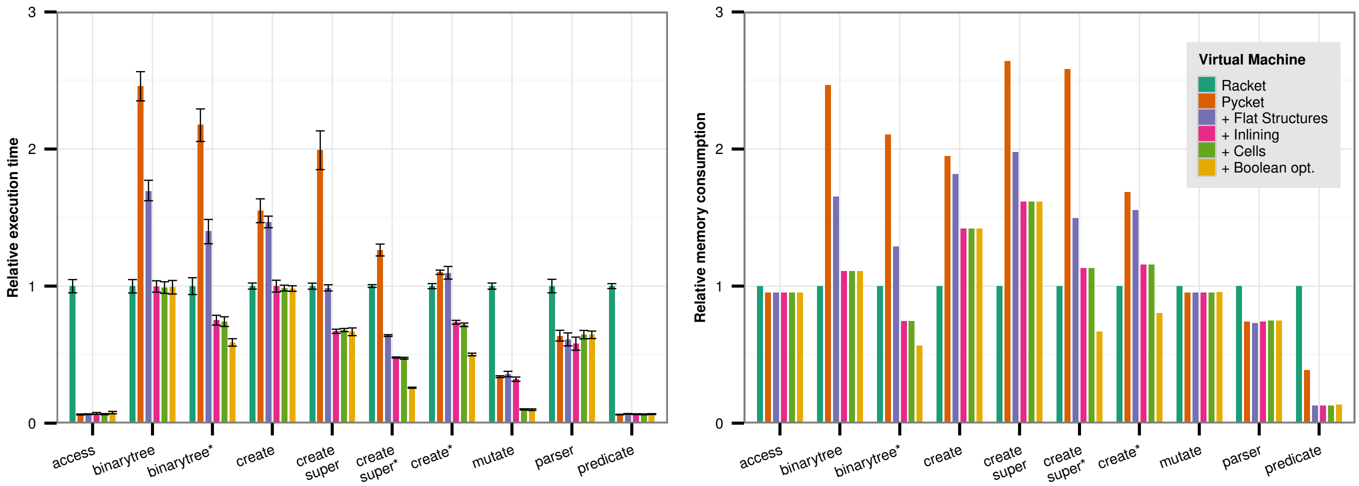

Figure 8: Benchmark results with execution times (left) and memory consumption (right) normalized to Racket. Lower is better.

In some benchmarks, such as *predicate*, *access*, *mutate*, and also *parser*, Pycket shows outright better execution time and memory consumption results, even without any optimization ("Pycket").

Expectedly, benchmarks that require the creation of many structures initially show worse performance, for example both *create* and both *binarytree* benchmarks.

6.2.2 General Optimizations

6.2.2.1 Flat Structure.

This optimization improves performance when the benchmarks frequently create structure instances, for example in all *create*…and *binarytree*…benchmarks. The impact on the remaining tests is less pronounced. Some benchmarks with intensive access operations show slightly worse performance results, for example *access*.

For memory consumption, the flat structure optimization should save $n \cdot (6 \cdot 0.319 + 1)$ words for n structures on average, according to Equation 1. However, in our tests, structures either always have a one super-type or do not have super-types at all. Thus, this formula transforms to $n \cdot (6+1) = n \cdot 7$ for structures with exactly one super-type, for example in *create/super* and *binarytree* and n for the *create* micro-benchmark where structures are created without super-type. Having one machine-word equals 64 bit and 15 000 000 structures in the *create* benchmark results in

$$n = 15\,000\,000 \cdot 64\,\text{bit} \approx 114.4\,\text{MB}.$$

The *create* benchmark shows a benefit of 115 MB. The predicted gain of memory consumption for the *create/super* benchmark with 30 000 000 structure instances is

$$n \cdot 7 = 30\,000\,000 \cdot 7 \cdot 64\,\text{bit} \approx 1602.2\,\text{MB}$$

which is approximately equal to the result of 1603.8 MB. For *binarytree*, whose nodes always have one super-type except the leaves, the result for trees of depth 22 is

$$n_1 \cdot 7 + n_2 = (2^{22} \cdot 7 + 2^{23}) \cdot 64\,\text{bit} \approx 288\,\text{MB}$$

which corresponds with test results of 288.7 MB. The the overall level of memory consumption for Pycket is bigger than for Racket.

As expected, benchmarks with an intensive creation of structures require much less memory. Other benchmarks, for example *mutate*, that do not create a hight number of structures, do not gain benefits in memory consumption from this optimization. ("+ Flat Structures")

6.2.2.2 Inlining.

As expected, all benchmarks except *mutate* gained execution time performance, especially for creation heavy benchmarks, where the avoided indirection shows in reduced execution time and memory consumption. The actual memory saving, according to Equation 2, should be $n \cdot 3 - \text{sizeOfSpecializedClasses}$ words, for n structure instances. The size of the specialized classes turned out to be insignificantly low. Having 15 000 000 structure instances in the *create* micro-benchmark, we should save

$$n \cdot 3 = 15\,000\,000 \cdot 3 \cdot 64\,\text{bit} \approx 343.3\,\text{MB}.$$

The measurement 344.6 MB differs only slightly. For the *create/super* micro-benchmark with 30 000 000 instances, we should save

$$n \cdot 3 = 30000000 \cdot 3 \cdot 64\,\text{bit} \approx 686.6\,\text{MB}.$$

Our measurement deviates slightly with 689.8 MB. Finally, for *binarytree* with a 22-level deep tree, we expected

$$n \cdot 3 = 2^{23} \cdot 3 \cdot 64\,\text{bit} \approx 192\,\text{MB}$$

which fits our measurement of 192.9 MB. ("+ Inlining")

6.2.2.3 Mutability separation with cells.

The *mutate* benchmark achieves a significant speed-up from the cell optimization, as the JIT can now treat the actual structure instance as immutable; the additional indirection pays off. As expected, other performance results remain approximately the same.

The Pycket automatic unboxing for small structures has *not* been enabled for this optimization level, hence, there is only minor influence of using cells on memory consumption on itself. ("+ Cells")

6.2.3 *Immutable Boolean Field Elision*

All benchmarks with IBFs — that is *create**, *create/super** and *binarytree** — achieve a speed-up and reduced memory consumption. In these particular benchmarks, the execution time becomes about 30 % faster. Memory savings range from 25 % to 40 %. At the same time, all other benchmarks are virtually untouched, showing next to no disadvantages of employing IBFE. ("+ Booleans opt.")

The actual memory saving, according to Equation 4, should be $n \cdot 2.3 \cdot 0.26 \cdot 0.7 \cdot 0.85 - sizeOfSpecializedClasses \approx 0.35n$ words, for n structure instances. However, the benchmarks deviate from the average numbers in the sense that the use of IBFs is well known and the three boolean-related benchmarks yield different but expected results, Also Pycket's automatic unboxing of small structure applies (cf. section 5.4) . The size of the specialized classes turned out to be insignificantly low.

6.2.3.1 *create*.*

Structures have one IBF per (two-field) instance, always being *false*, all fields immutable, yielding $n \cdot 2 \cdot \frac{1}{2} = n$. Considering Pycket's automatic unboxing, *create** makes use of Racket's a *fixnum* for the second structure field. Hence, compared with the cells optimization level, additional two words per structure are saved, yielding $n \cdot (2 + 2 \cdot \frac{1}{2}) = 3n$. Having 15 000 000 structure instances for *create**, we should save

$$3n = 3 \cdot 15000000 \cdot 64\,\text{bit} \approx 343.3\,\text{MB}.$$

The measured result 344.6 MB differs only slightly.

6.2.3.2 *create/super*.*

Structures have two IBF per (three-field) instance, always being *false*, all fields immutable, yielding $n \cdot 3 \cdot \frac{2}{3} = 2n$. However, the third field being a Racket *fixnum*, and the number of *actual* fields dropping from three to one due to IBFE, Pycket's unboxing applies, and additional two words will be saved per structure instance, eventually yielding $n \cdot (2 + 3 \cdot \frac{2}{3}) = 4n$. Having 30 000 000 structure instances *create/super**, we should save

$$4n = 4 \cdot 30000000 \cdot 64\,\text{bit} \approx 915.5\,\text{MB}.$$

The measured result 881.8 MB deviates less than 4 %.

6.2.3.3 *binarytree*.*

Structures have one IBF with a *false* per instance, yielding n. With a tree depth of 22, we should save

$$n = 2^{23} \cdot 64\,\text{bit} = 64\,\text{MB}$$

witch matches the measured result exactly.

6.3 Limitations

We only evaluated the efficiency of structures in Pycket on self-written benchmarks. Although they are well suited to test performance of basic operations with structures, real-world applications may show different behavior as part of future work. Once feasible, more elaborate benchmarks will be used.

JIT warm-up time has an impact on execution time. We use our benchmarks with a sufficient warm-up time, which is not guaranteed to be always reachable in real-world applications. Also, warm-up time may differ between benchmarks. In order to illustrate the importance of the sufficient warm-up time in micro-benchmarks, we ran the *create/super* micro-benchmark with different numbers of iterations. The results of this benchmark are presented in Figure 9. Pycket shows pure performance results with a small number of iterations, but starting with some sufficient number (about 30 millions in this particular micro-benchmark), Pycket is continuously faster. The slowness of Pycket at the beginning arise from the JIT warm-up. Therefore, we use different, sufficiently large numbers of iterations in every benchmark to show the well-established performance.

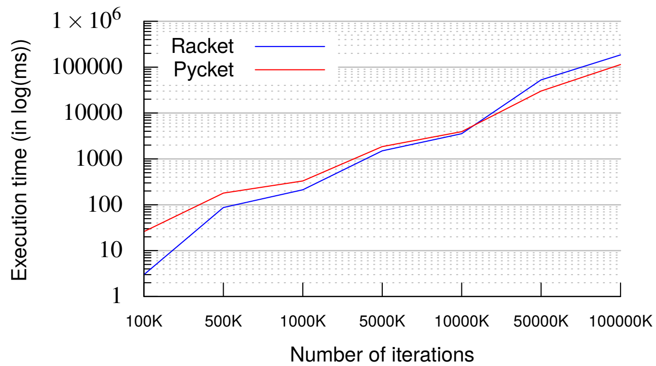

Figure 9: Execution times (in log(ms)) of *create/super* micro-benchmarks for Racket and Pycket with different number of iterations illustrate the influence of JIT warm-up. Lower is better.

Finally, we are unable to influence internal CPU optimizations, such as enabling a boost-mode. However, such optimizations should work same for both Racket and Pycket running single threaded.

7. RELATED WORK

Late Data Layout is a lightweight annotations mechanism [22] to eliminate limitations of coercions between internal data representations. Boxing and unboxing operations are not inserted eagerly by a compiler but only at execution time, with checks that ensure the consistency of the data representation. The checks are based on multi-phase *type-driven* data representation transformations local-type inference. Hence, unnecessary transformation operations can be omitted and data-type representations are added optimally.

The object storage model [23] of Truffle [24] creates every object as an instance of a *storage class*, which works as a container for the instance data. This class references a *shape* that describes the object's data format and behavior. Shapes and all their accessible data are immutable, but the reference to a shape from the storage class themselves can vary over time. Thus, any change of the object's shape results in a new shape. The proposed approach is suitable for sufficiently efficient compilation with further optimizations, such as polymorphic inline cache (PIC) for efficient object's property lookup.

A more specialized approach to increase performance of data structures in VMs is *storage strategies* [5] for collections of homogeneously typed elements. If possible, they are stored unboxed and their type is stored separately and only once with a special object called *strategy*. For example, adding an integer to an empty collection enables the *integer* strategy for this collection and this integer and all subsequent integers will be saved unboxed. However, adding a non-integers, for example a string, causes a transition to a generic strategy, because the collection is now heterogeneous. It is assumed that such transformations are unlikely, which is shown by the authors. A similar approach is used for structures with mutable cells in this work. Every cell has its strategy and its values are saved unboxed, unless under a generic strategy.

While pointer tagging and strategies reduce memory consumption by unboxing values, it is also possible to reduce the size of the structure itself, when a substantial amount of structures is allocated. *Structure vectors* group structures of the same type, allowing to store the header and the type descriptor only once [9]. This optimization is most beneficial when large amounts of structures are used, achieving a speed-up of up to 15 %. Yet, while allocation becomes faster, field access and especially type descriptor access become up to three to four times slower [9]. However, the allocation of a big number of structures is not very common in Racket (cf. section 3).

An effective run-time representation exists for R^6RS Scheme records [16] where each record has an associated execution time representation, record-type descriptor (RTD), determining its memory layout. When an RTD is created, the compiler calculates record sizes and field offsets for this record type similar to the way presented here. They have flat representation with inlined fields, quite similar to structures Pycket. A special interface allows to store raw integers, untagged floating point numbers, and raw machine pointers, in addition to ordinary Scheme data types.

The representation of structures in Racket's implementation is related to our work, too. However, we deliberately chose to not investigate the implementation but rather base our approach solely on the extensive documentation and the static and dynamic analyses. A comparison of our implementation to Racket's is part of future work.

8. CONCLUSION AND FUTURE WORK

We presented an analysis of record structure usage in Racket and proposed optimizations that are fit for an efficient implementation. We considered three common approaches and devised a novel optimization for immutable boolean fields. We applied these approaches to Pycket, a tracing-JIT-based implementation of Racket, and achieve a significant speed-up compared to Racket in provided micro-benchmarks with a sufficient warm-up time. We evaluated the impact of our optimizations with a set of micro-benchmarks.

Our results suggest further investigation of *unboxing* values, as homogenised fields in structures make up about 85 % in Racket on average. *Adaptive optimizations* [19] show promising initial results and may be applied to records in the future. Finally, once Pycket's feature coverage is sufficient, we will run a broader range of benchmarks.

9. APPENDIX: COMPREHENSIVE BENCHMARK RESULTS

The results of all benchmarks are presented in Table 2 (execution time) and Table 3 (memory consumption). The first rows of these tables contain Racket numbers for references. The second row present the unoptimized implementation. All subsequent rows represent improvements with each optimization, in an accumulated fashion, that is, the last row represents Pycket with all optimizations presented here. Benchmarks annotated with * make explicit use of booleans. All error values are bootstrapped [10] confidence intervals for a 95 % confidence level.

Acknowledgments

We gratefully acknowledge the financial support of HPI's Research School and the Hasso Plattner Design Thinking Research Program (HPDTRP). We want to thank Carl Friedrich Bolz for fruitful discussions, Spenser Bauman, Jeremy Siek, and Sam Tobin-Hochstadt for their support during implementation, and the anonymous reviewers for their insightful comments. Carl Friedrich Bolz is supported by the EPSRC *Cooler* grant EP/K01790X/1.

10. REFERENCES

[1] D. Ancona, M. Ancona, A. Cuni, and N. Matsakis. "RPython: A Step Towards Reconciling Dynamically and Statically Typed OO Languages". In: *Proc. of DLS 2007*. DLS '07. Montreal, Quebec, Canada: ACM, 2007, pp. 53–64.

[2] D. F. Bacon, S. J. Fink, and D. Grove. "Space- and time-efficient implementation of the Java object model". In: *ECOOP 2002 — Object-Oriented Programming*. Ed. by B. Magnusson. Vol. 2374. LNCS. Springer, 2002, pp. 111–132.

[3] S. Bauman, C. F. Bolz, R. Hirschfeld, V. Krilichev, T. Pape, J. Siek, and S. Tobin-Hochstadt. "Pycket: A Tracing JIT For a Functional Language". In: *Proc. of ICFP 2015*. ICFP '15. Vancouver, British Columbia, Canada: ACM, 2015.

[4] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. "Tracing the meta-level: PyPy's tracing JIT compiler". In: *Proc. of ICOOOLPS 2009*. ACM. 2009, pp. 18–25.

[5] C. F. Bolz, L. Diekmann, and L. Tratt. "Storage strategies for collections in dynamically typed languages". In: *Proc. of OOPSLA 2013*. ACM. 2013, pp. 167–182.

[6] C. F. Bolz, A. Kuhn, A. Lienhard, N. D. Matsakis, O. Nierstrasz, L. Renggli, A. Rigo, and T. Verwaest. "Back to the future in one week—implementing a Smalltalk VM in PyPy". In: *Self-Sustaining Systems*. Springer, 2008, pp. 123–139.

[7] C. F. Bolz, M. Leuschel, and D. Schneider. "Towards a Jitting VM for Prolog Execution". In: *Proc. of PPDP 2010*. PPDP '10. Hagenberg, Austria: ACM, 2010, pp. 99–108.

[8] C. F. Bolz, T. Pape, J. Siek, and S. Tobin-Hochstadt. "Meta-tracing makes a fast Racket". In: *Workshop on Dynamic Languages and Applications*. 2014.

[9] B. Cérat and M. Feeley. "Structure Vectors and their Implementation". In: *Scheme and Functional Programming Workshop*. 2014.

[10] A. C. Davison and D. V. Hinkley. In: *Bootstrap Methods and Their Application*. Cambridge, 1997. Chap. 5.

[11] M. Felleisen and D. P. Friedman. *Control Operators, the SECD-machine, and the λ-calculus*. Indiana University, Computer Science Department, 1986.

[12] M. Flatt. *Reference: Racket*. Tech. rep. PLT-TR-2010-1. PLT Design Inc., 2010.

[13] M. Flatt, R. Findler, and M. Felleisen. "Scheme with Classes, Mixins, and Traits". In: *Programming Languages and Systems*. Ed. by N. Kobayashi. Vol. 4279. LNCS. Springer, 2006, pp. 270–289.

[14] R. P. Gabriel. "LISP: Good news, bad news, how to win big." In: *AI EXPERT*. 6.6 (1991), pp. 30–39.

Table 2: Execution times (in ms) for Racket and Pycket (without optimizations, with flat structures, with inlined fields, with cells, and with IBFE). Less is better.

VM / Optimization	Create	Create*	Create/sup.	Create/sup.*	Predicate	Access	Mutate	Binary tree	Bin. tree*	Parser
Racket	4982 ± 134	5210 ± 114	19 684 ± 726	20 243 ± 97	3585 ± 105	2917 ± 125	4306 ± 223	1817 ± 61	2046 ± 82	1061 ± 52
Pycket	7027 ± 39	5683 ± 134	35 779 ± 1395	24 657 ± 772	221 ± 8	172 ± 16	1214 ± 14	4735 ± 202	3959 ± 205	715 ± 61
+ Flat structure	6116 ± 90	5245 ± 140	20 575 ± 742	14 132 ± 160	227 ± 10	162 ± 5	1291 ± 88	3133 ± 126	2379 ± 132	732 ± 54
+ Inlined fields	4821 ± 169	4002 ± 122	14 682 ± 261	10 654 ± 104	226 ± 11	177 ± 10	1250 ± 23	1976 ± 136	1429 ± 76	667 ± 23
+ Cells	4886 ± 70	3894 ± 58	14 446 ± 488	10 066 ± 8	224 ± 21	171 ± 6	355 ± 12	1850 ± 115	1504 ± 29	684 ± 53
+ Booleans	4726 ± 94	2586 ± 77	14 317 ± 432	5517 ± 81	216 ± 14	161 ± 5	387 ± 15	2016 ± 109	1224 ± 51	666 ± 28

Table 3: Memory consumption (in MB) for Racket and Pycket (without optimizations, with flat structures, with inlined fields, with cells, and with IBFE). Less is better.

VM / Optimization	Create	Create*	Create/sup.	Create/sup.*	Predicate	Access	Mutate	Binary tree	Bin. tree*	Parser
Racket	871.7 ± 0.0	871.7 ± 0.1	1923.9 ± 0.0	1924.0 ± 0.0	50.5 ± 0.7	813.8 ± 0.0	813.8 ± 0.0	376.3 ± 0.7	376.6 ± 0.1	52.1 ± 0.0
Pycket	1692.5 ± 0.0	1462.5 ± 0.0	5365.0 ± 4.2	4882.6 ± 3.9	6.5 ± 0.0	769.6 ± 0.0	769.7 ± 0.0	875.7 ± 0.1	747.2 ± 0.1	33.3 ± 0.2
+ Flat structure	1577.4 ± 0.1	1347.6 ± 0.0	3761.3 ± 0.0	2841.6 ± 0.0	6.5 ± 0.0	769.5 ± 0.0	769.7 ± 0.0	587.0 ± 0.1	457.7 ± 0.1	34.7 ± 0.5
+ Inlined fields	1232.7 ± 0.0	1003.0 ± 0.0	3071.5 ± 0.0	2152.1 ± 0.0	6.5 ± 0.0	769.5 ± 0.0	769.7 ± 0.0	394.1 ± 0.1	265.1 ± 0.1	34.1 ± 0.3
+ Cells	1232.8 ± 0.1	1003.0 ± 0.0	3071.5 ± 0.0	2152.1 ± 0.0	6.5 ± 0.0	769.5 ± 0.0	769.7 ± 0.0	394.1 ± 0.1	265.1 ± 0.1	34.6 ± 0.3
+ Booleans	1233.0 ± 0.1	696.1 ± 0.0	3071.8 ± 0.1	1270.3 ± 0.0	6.7 ± 0.0	769.8 ± 0.1	769.9 ± 0.0	394.1 ± 0.1	201.1 ± 0.1	34.9 ± 0.4

[15] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. "Back to the future: the story of Squeak, a practical Smalltalk written in itself". In: *ACM SIGPLAN Notices*. Vol. 32. 10. ACM. 1997, pp. 318–326.

[16] A. W. Keep and R. Dybvig. "A run-time representation of scheme record types". In: *J Funct Program* 24 (Special Issue 06 2014), pp. 675–716.

[17] R. Mitchell, J. McKim, and B. Meyer. *Design by contract, by example*. Addison Wesley, 2001.

[18] M. E. Noth. "Exploding Java Objects for Performance". PhD thesis. University of Washington, 2003.

[19] T. Pape, C. F. Bolz, and R. Hirschfeld. "Adaptive Just-in-time Value Class Optimization: Transparent Data Structure Inlining for Fast Execution". In: *Proc. of SAC 2015*. Vol. 2. SAC '15. Salamanca, Spain: ACM, 2015.

[20] A. Rigo and S. Pedroni. "PyPy's approach to virtual machine construction". In: *Proc. of OOPSLA 2006*. Portland, Oregon, USA: ACM, 2006, pp. 944–953.

[21] G. J. Sussman and G. L. Steele Jr. "Scheme: A interpreter for extended lambda calculus". In: *Higher-Order and Symbolic Computation* 11.4 (1998).

[22] V. Ureche, E. Burmako, and M. Odersky. "Late data layout: unifying data representation transformations". In: *Proc. of OOPSLA 2014*. ACM. 2014, pp. 397–416.

[23] A. Wöß, C. Wirth, D. Bonetta, C. Seaton, H. Mössenböck, and C. Humer. "An Object Storage Model for the Truffle Language Implementation Framework". In: *Proc. of PPPJ 2014*. PPPJ '14. Cracow, Poland: ACM, 2014, pp. 133–144.

[24] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. "One VM to rule them all". In: *Proc. Onward! 2013*. ACM. 2013, pp. 187–204.