A Sufficiently Smart Compiler for Procedural Records

Andrew W. Keep R. Kent Dybvig
Cisco Systems, Inc. and Indiana University
{akeep,dyb}@cisco.com

Abstract

Many languages include a syntax for declaring programmer-defined structured data types, i.e., structs or records. R6RS supports syntactic record definitions but also allows records to be defined procedurally, i.e., via a set of run-time operations. Indeed, the procedural interface is considered to be the primitive interface, and the syntactic interface is designed to be macro expandable into code that uses the procedural interface. Run-time creation of record types has a potentially significant impact. In particular, record creation, field access, and field mutation cannot generally be open coded, as it can be with syntactically specified records. Often, however, the shape of a record type can be determined statically, and in such a case, performance equivalent to that of syntactically specified record types can be attained. This paper describes an efficient run-time implementation of procedural record types, discusses its overhead, and describes a set of compiler optimizations that eliminate the overhead when record-type information can be determined statically. The optimizations improve the performance of a set of representative benchmark programs by over 20% on average.

1. Introduction

The record system for the Revised^6 Report on the Algorithmic Language Scheme (R6RS) [16] differs from more traditional methods of defining structured data types, such as structs in C, in that record types, record predicates, record constructors, field accessors, and field mutators are created dynamically, i.e., at run time.

A program creates a record type by creating a *record-type descriptor* (RTD) that describes the type. When creating an RTD, the program can specify a parent RTD, in which case the resulting record type is an extension of the parent record type via single inheritance. The program must specify the fields that each instance of the type should have, in addition to those of its parent type, if any, and whether each is mutable. It must also specify whether the record type is *sealed* and whether the record type is *opaque*. A sealed record type cannot be used as the parent type for another record type, i.e., it cannot be extended. If a record type is opaque, instances of the type cannot be inspected (Section 2.2). The program can also supply a UID, in which case the record type is considered *nongenerative* rather than *generative*. Any subsequent attempt to create a record type with the same UID produces the same record type, except that if the parent, fields, sealedness, or opaqueness are not the same, the attempt causes an exception to be raised. Each record type is distinct from all built-in types and from other record types, although an instance of a record type is considered an instance of its parent type, if any.

Once an RTD has been created, the program can use it to create a record predicate, which returns true only for instances of the record type, along with accessors and mutators that can be used to extract values from or store values into the fields of an instance of the record type. It can also create a record constructor for the record type, but it must first create a *record-constructor descriptor* (RCD).

An RCD controls how direct instances of the record type are created and partially controls how instances of its child types are created. When creating an RCD, a program specifies a parent RCD, if the record type has a parent type, and a *protocol*. The protocol determines the arguments expected by the record constructor, how the fields of a new instance are initialized from the arguments, and possibly actions to be taken on the instance before it is returned from the constructor.

The RTD, RCDs, constructors, predicate, accessors, and mutators for a record type can be created either via a set of run-time procedures that comprise a *procedural interface* or via a *syntactic interface*. Uses of the syntactic interface can always be expanded into uses of the procedural interface, which is the simplest way to implement the syntactic interface.

While the syntactic interface is often more convenient, the procedural interface is more flexible. For example, it allows the construction of portable interpreters that allow the programs they interpret to construct host-compatible record types. It also allows the creation of arbitrary new data types at run time, e.g., based on data read from an input file. When coupled with the *inspection interface* (Section 2.2), which includes a procedure to extract the RTD from a record instance, it allows the construction of portable debuggers and pretty-printers for records.

The flexibility of the procedural interface and the simplicity of implementing the syntactic interface via the procedural interface comes at a price. Because RTDs, RCDs, constructors, predicates, accessors, and mutators are all created at run time, record creation, access, and mutation are out-of-line operations requiring indirect jumps and possible pipeline stalls, possibly along with memory overhead beyond that which would be necessary in a straightforward implementation of a syntactic interface, e.g., to extract the RTD (for type checking) and field index from the closure of an accessor or mutator. In addition, record-definition code must be put at the top level of a program or library if creation of the RTDs, RCDs, constructors, predicates, accessors, and mutators is to happen only at initialization time—an unfortunate constraint and not an obvious one when the syntactic interface is used.

This paper describes an efficient implementation of the procedural interface and a set of compiler optimizations that eliminate its inherent overhead when sufficient information about a record type can be determined statically. In particular:

- RTDs for nongenerative record types are created at compile time, and RCDs are dropped entirely.
- Allocation is performed inline except when there are multiple constructor call sites and the code is too large to replicate.
- Record-predicate calls reduce to an inline record-type check.
- A record field reference or mutation reduces to an inline record-type check and a single memory operation, or just a single memory operation when the type of a record instance can be determined at compile time.

In addition, operations on nongenerative record types with immutable fields occur at compile time whenever possible, i.e., whenever the arguments to a constructor can be determined at compile time, and whenever a field accessor is applied to a constant record instance.

Sufficient information can always be determined statically when a record type is defined exclusively via the syntactic interface. Thus, the optimizations above always apply to the syntactic interface. Furthermore, it is rarely necessary to place a syntactic definition for a nongenerative record type at top level, since it does not involve any run-time overhead beyond evaluating the programmer-supplied protocol expression, if any.

The optimizations improve the running time of a set of benchmarks by an average of over 20% versus the base procedural interface and a few percent over an implementation of records simulated by vectors. For some programs, the optimizations are substantially more effective; for example, they improve the speed of the Chez Scheme compiler, which makes extensive use of records, by nearly 90%.

The rest of this paper is organized as follows. Section 2 describes the R6RS procedural, syntactic, and inspection interfaces. Section 3 describes the implementation of the procedural interface. Section 4 describes the compiler optimizations. Section 5 describes a set of benchmarks and compares how they run with the optimizations enabled or disabled. Section 6 discusses related work. Finally, Section 7 discusses future work and concludes.

2. The R6RS Record System

This section presents an overview of the procedural, syntactic, and inspection interfaces to the R6RS record system and can be skipped by readers who are already familiar with the record system.

We start by looking at how to create a simple `point` record type that contains `x` and `y` fields, using the procedural interface.

```
(define point-rtd
  (make-record-type-descriptor 'point #f #f
    #f #f '#((mutable x) (mutable y))))
(define point-rcd
  (make-record-constructor-descriptor
    point-rtd #f #f))
(define make-point (record-constructor point-rcd))
(define point? (record-predicate point-rtd))
(define point-x (record-accessor point-rtd 0))
(define point-x-set! (record-mutator point-rtd 0))
(define point-y (record-accessor point-rtd 1))
(define point-y-set! (record-mutator point-rtd 1))
```

The `point` record-type descriptor (RTD) specified here does not inherit from any other RTD. It is generative, so a new (and unique) RTD is created each time the `make-record-type-descriptor` call is evaluated. In particular, if the code appears in a loop, a new RTD is constructed on each iteration of the loop. The RTD is not sealed, meaning it can be a parent RTD to other record types. The RTD is not opaque, meaning the RTD can be inspected. Finally, the RTD specifies two mutable fields `x` and `y`.

Creating the constructor for the `point` record type requires a record-constructor descriptor (RCD). The RCD specified here is the default RCD. The default RCD specifies that there is no parent RCD and no protocol. It creates a constructor procedure that expects one argument for each field in the record and initializes each field to the value of the corresponding argument. The constructor for the `point` record type is generated with the RCD. The predicate, accessors, and mutators for `point` record type are generated with the RTD and, in the case of the accessors and mutators, the index of the field.

Most record types, including the `point` record type, can be created using the more convenient syntactic interface:

```
(define-record-type point
  (fields (mutable x) (mutable y)))
```

The `define-record-type` form can be implemented as a macro, and this occurrence might expand into the procedural-interface code shown above. The only difference, in this case, is that the `point-rtd` and `point-rcd` variables are not made visible via the syntactic interface, though the RTD and RCD are still created. (The syntactic interface does provide forms for looking up the RTD and RCD from a record name.) In addition to the `fields` clause, `define-record-type` can specify its generativity with the `nongenerative` clause, whether it is sealed with the `sealed` clause, whether it is opaque with the `opaque` clause, a parent record type with the `parent` or `parent-rtd` clauses, and a record protocol with the `protocol` clause.

2.1 Protocols and inheritance

Record inheritance in the R6RS record system is designed to allow child record types to be created and used without specific knowledge of parent fields. This is why field indices for a child record type begin at zero even when the parent record type has one or more fields. It is also why protocols exist. Protocols provide an expressive way to abstract record creation, but this ability could be simulated by embedding the creation procedure within another procedure. More importantly, they free code that allocates a child record type from needing to know about the parent's fields, how to initialize them, and other actions required when an instance of the parent record type is created. The code needs to know only the arguments expected by the parent protocol. This is especially convenient when a parent record type is modified during program development, since code that deals with child record types need not change as long as the parent protocol remains the same.

A protocol is analogous to a constructor or init method in object-oriented programming and serves a similar purpose, with an appropriately more functional style. One use for a record protocol is to specify default values for fields that do not need to be specified when an instance of the record is created. Protocols are not limited to this, though, and can be used, e.g., to check for valid constructor arguments, register records with some global database, or mutate the record being constructed to create cycles.

For instance, imagine we want to create a record that stores a color name and the red, green, and blue (RGB) values for the color. We could create a record like the following:

```
(define-record-type color (fields name r g b))
```

The `color` record type is generative, is not sealed, is not opaque, and has four immutable fields: `name`, `r`, `g`, and `b`. In our representation, we always want `name` to be a symbol and `r`, `g`, and `b` to be integer numbers between 0 and 255. This restriction could be enforced through a protocol as follows:

```
(define-record-type color
  (fields name r g b)
  (protocol
    (lambda (new)
      (lambda (name r g b)
        (unless (symbol? name)
          (error 'make-color
            "name must be a symbol" name))
        (unless (and (fixnum? r) (<= 0 r 255)
                     (fixnum? g) (<= 0 g 255)
                     (fixnum? b) (<= 0 b 255))
          (error 'make-color
            "RGB value outside range" r))
        (new name r g b)))))
```

The `protocol` keyword specifies that this object has a protocol expression. Because the `color` record type does not inherit from any other record types, its protocol is passed a procedure to build the record that accepts one argument for each field in `color`. In this case, the protocol expression returns a procedure with one argument for each field in the `color` record type. This procedure performs the checks and then creates a new `color` record instance.

Another feature we might like is to store each color record in a global list of colors. If the record definition for `color` is at the top-level of a library or program, this will work without any changes. However, since the record type is generative, if it was moved into a context where it could be executed more then once, it might lead to the global list containing instances of many `color` record types with the same shape, but different RTDs. To avoid this we mark `color` nongenerative.

```
(define *colors* '())

(define-record-type color
  (nongenerative)
  (fields name r g b)
  (protocol
    (lambda (new)
      (lambda (name r g b)
        (unless (symbol? name)
          (error 'make-color
            "name must be a symbol" name))
        (unless (and (fixnum? r) (<= 0 r 255)
                     (fixnum? g) (<= 0 g 255)
                     (fixnum? b) (<= 0 b 255))
          (error 'make-color
            "RGB value outside range" r))
        (let ((c (new name r g b)))
          (set! *colors* (cons c *colors*))
          c)))))
```

With this definition of the `color` record type, each new color is recorded in a global list of colors. Specifying colors by RGB value is convenient, but if we are working on a web project, we might want to create colors that store the HTML hexadecimal representation as well as the RGB values. We can derive a new `web-color` record type from `color` to accomplish this.

```
(define-record-type web-color
  (parent color)
  (fields hex-color)
  (nongenerative)
  (protocol
    (lambda (pargs->new)
      (lambda (name r g b hex-color)
        (let ((new (pargs->new name r g b)))
          (new hex-color))))))
```

The `web-color` record type indicates that it inherits from `color` by naming `color` in the `parent` clause. It also provides a protocol. The first difference evident in the `web-color` protocol is that instead of being passed a procedure that takes all of the fields for the parent and the child record, it receives a procedure in the `pargs->new` formal that expects the parent arguments and returns a procedure that expects the child field values (in this case, just the value of `hex-color`). When a new `web-color` is created it should be added to the global `*colors*` list, just as a new `color` record is added. Fortunately, the protocol for `color` is invoked as part of the process of creating the new `web-color` so there is no need for the `web-color` protocol to duplicate this effort.

The `color` protocol checks the arguments and adds the record to the `*colors*` list, but we might not want the user of the `web-color` record type to need to supply the color twice, once as a web color and once as a set of RGB values. In fact, this could lead to inconsistencies, since the web color string and the RGB value might specify different colors. One option is to use a protocol to check that these are consistent, but a better option would be to perform the conversion. Again, we can use a protocol to do the work, with an appropriately defined `rgb->hex-color` (not shown):

```
(define-record-type web-color
  (parent color)
  (fields hex-color)
  (nongenerative)
  (protocol
    (lambda (pargs->new)
      (lambda (name r g b)
        ((pargs->new name r g b)
         (rgb->hex-color r g b))))))
```

The protocol specified above converts from an RGB representation to a hexadecimal string representation used by web pages to indicate colors. This value is then filled in for the `hex-color` field of the `web-color` record type.

Now that we have a set of record types defined, we define some color records, find the color named `red` in the color list, and extract its hex-color value:

```
(make-web-color 'red 255 0 0)
(make-web-color 'green 0 255 0)
(make-web-color 'blue 0 0 255)

(define red
  (find
    (lambda (c) (eq? (color-name c) 'red))
    *colors*))

(web-color-hex-color red)  ⇒  "#FF0000"
```

The protocol for the `web-color` record types calculates the HTML hexadecimal color from the red, green, and blue arguments. The protocol for the `color` record type checks the name is a symbol

and that the red, green, and blue values are within the appropriate range.

2.2 The inspection interface

It is also possible to determine information about the type of a record instance using the inspection interface, as the following example demonstrates.

```
(define red-rtd (record-rtd red))
(record-type-descriptor? red-rtd)  ⇒  #t
(record-type-name red-rtd)         ⇒  web-color
(record-type-generative? red-rtd)  ⇒  #f
(record-type-uid red-rtd)          ⇒  web-color
(record-type-sealed? red-rtd)      ⇒  #f
(record-type-opaque? red-rtd)      ⇒  #f
(record-field-mutable? red-rtd)    ⇒  #f
(record-type-field-names red-rtd)  ⇒  #(hex-color)
(define p-rtd
  (record-type-parent red-rtd))
(record-type-name p-rtd)           ⇒  color
```

These operations allow a program (e.g., a portable debugger) to retrieve information about the record type, and when combined with the procedural interface for creating record accessors and mutators, also allow the program to retrieve or even modify the contents of the record instance. If the record type of a record instance is opaque, the `record-rtd` procedure raises an exception, effectively preventing inspection of the record and its type.

3. Implementation

Although we hope that most uses of the procedural record system can be optimized into efficient compiled code, it is still important to have an efficient implementation of the procedural interface. There will always be cases where the shape of a record type cannot be determined at compile time, and the compiler optimizations cannot be applied, as when record types are generated based on information available only at run time. An efficient implementation of the procedural interface also gives us a competent base against which to measure the benefits of our optimizations when the optimizations can be applied.

The implementation precalculates record sizes and field offsets to avoid repeating these calculations each time a record instance is created or a field is accessed or mutated. It also special-cases the creation of record constructors with just a few arguments to avoid the use of "rest" argument lists and `apply`.

The first step in creating a record type is to create the record-type descriptor (RTD). The `make-record-type-descriptor` procedure begins by checking that the newly specified record type is valid. If the record type is nongenerative, this includes checking to see if the RTD already exists. If the RTD exists, the arguments are checked to ensure the RTD matches those specified. If the RTD does not exist or the record type is generative a new RTD must be created. If a parent RTD is specified, the inheritance hierarchy is traversed to gather the full field list for the new record type. Once the full field list is known, the total size of the record type and the byte offset for each field is calculated. Information about each field is stored in a vector with the field name, a flag to indicate if the field is mutable, and the byte-offset of the field[1]. The fields list, along with the parent RTD, record type name, record type UID, sealed flag, and opaque flag are all stored in the RTD. Our implementation uses a record to represent the RTD. The RTD's own RTD is also a record, with a base RTD ultimately terminating the chain.

The `make-record-constructor-descriptor` procedure is used to create a record-constructor descriptor (RCD) from an RTD. It first checks that it is given a valid RTD. When a parent RCD is specified, it also checks that it is an RCD for the parent RTD, and that a protocol procedure is also specified. The RCD is represented by a record and contains the RTD, parent RCD, and protocol.

Generating the record constructor procedure is one of the more complicated parts of implementing the record system. The combination of protocols and inheritance means that the constructor must first gather up the values for all of the fields by calling each protocol procedure in the hierarchy in turn, then construct the new record instance, and finally return the record instance through each protocol procedure. The implementation for this is shown in Figure 1.

First, a procedure for constructing the final record instance is defined as `rc`. If the number of fields is small (six or fewer), it creates this procedure with an exact number of arguments and then calls the internal record constructor `$record` with the RTD and the values of the fields. If the number is larger then six, it defaults to using a rest argument and then calls `$record` with apply. This procedure also checks to make sure the correct number of arguments are received. If the RCD does not specify a protocol, `rc` is the constructor. Otherwise, the protocol procedure must be called. If no parent RTD is specified, the `rc` is passed to the protocol as the procedure to construct the final record. If a parent RTD is specified, the procedure must traverse each level of the inheritance hierarchy until it reaches the base of the inheritance or finds a parent that does not specify a protocol. At each stage of this recursion the parent protocol is called on a constructed procedure until values for all the fields in the record type are gathered and the record instance can be constructed. The internal record creation primitive `$record` is responsible for allocating and filling the fields.

The `record-accessor` procedure takes an RTD and an index into the field vector constructed by `make-record-type-descriptor`. It uses the `find-fld` helper to find the field offset as shown in Figure 2. It then constructs the accessor procedure using the field RTD to check that the argument to the accessor is a valid record instance and the precalculated field offset to reference the field. The type check uses an extended version of `record?` that takes an RTD as a second argument. The `find-fld` helper checks that the first argument to `record-accessor` is an RTD and that the index is a non-negative fixnum. The list of fields is then retrieved from the RTD, and if there is a parent RTD, the index is adjusted to select the correct field from the child fields. The field offset is precalculated by `make-record-type-descriptor` so it is simply retrieved from the field descriptor in the RTD with the `fld-byte` accessor. The accessor procedure returned has the RTD and the offset in its closure, so it does not need to calculate it when it is called.

The `record-mutator` procedure requires much the same information as the `record-accessor` procedure. It uses the `find-fld` helper to find the field, but must perform an additional check to ensure the field is mutable before returning the mutator procedure. It also replaces the `mem-ref` operation of the `record-accessor` with a `mem-set!`. The implementation of this is in Figure 3.

The `record-predicate` procedure uses the RTD to determine if it should call the built-in `$sealed-record?` predicate or the `record?` predicate for the RTD. Its implementation is shown in Figure 4. The `$sealed-record?` predicate is potentially faster,

[1] Chez Scheme allows a native type, such as `integer-8` or `double` to be specified for a field type, and this information is also stored in the field list, though for R6RS records, the type is always Scheme object.

```
(define record-constructor
  (lambda (rcd)
    (unless (rcd? rcd) (error 'record-constructor "not a record constructor descriptor" rcd))
    (let ((rtd (rcd-rtd rcd)) (protocol (rcd-protocol rcd)) (flds (rtd-flds rtd)))
      (let ((nflds (length flds)))
        (define rc
          (case nflds
            ((0) (lambda () ($record rtd)))
            ((1) (lambda (t0) ($record rtd t0)))
            ((2) (lambda (t0 t1) ($record rtd t0 t1)))
            ((3) (lambda (t0 t1 t2) ($record rtd t0 t1 t2)))
            ((4) (lambda (t0 t1 t2 t3) ($record rtd t0 t1 t2 t3)))
            ((5) (lambda (t0 t1 t2 t3 t4) ($record rtd t0 t1 t2 t3 t4)))
            ((6) (lambda (t0 t1 t2 t3 t4 t5) ($record rtd t0 t1 t2 t3 t4 t5)))
            (else (lambda xr
                    (unless (fx=? (length xr) nflds) (error #f "incorrect number of arguments" rc))
                    (apply $record rtd xr)))))
        (if protocol
            (protocol
              (cond
                ((rtd-parent rtd) =>
                 (lambda (prtd)
                   (lambda pp-args
                     (lambda vals
                       (let f ((prcd (rcd-prcd rcd)) (prtd prtd) (pp-args pp-args) (vals vals))
                         (apply
                           (cond
                             ((and prcd (rcd-protocol prcd)) =>
                              (lambda (protocol)
                                (protocol
                                  (cond
                                    ((rtd-parent prtd) =>
                                     (lambda (prtd)
                                       (lambda pp-args
                                         (lambda new-vals
                                           (f (rcd-prcd prcd) prtd pp-args (append new-vals vals))))))
                                    (else (lambda new-vals (apply rc (append new-vals vals))))))))
                             (else (lambda new-vals (apply rc (append new-vals vals)))))
                           pp-args))))))
                (else rc)))
            rc)))))
```

Figure 1. The `record-constructor` procedure

since it can perform a simple `eq?` check between the RTD in the record instance and the RTD supplied. The `record?` predicate must take into account the possibility of inheritance. The record predicate returned has only the RTD in its closure.

This implementation, however, is about as efficient as we can make it without compiler optimizations such as those described in the following section. Record accessors and mutators use precalculated indices. Constructors for smaller records are special-cased to avoid some rest-list and `apply` overhead.

Yet, many sources of overhead remain. A constructor, predicates, accessor, or mutator created returned by the procedural interface must be called via an anonymous call involving extraction of a code pointer from the procedure and an indirect jump that together result in additional memory traffic and a potential pipeline stall. A record predicate, accessor, or mutator must also extract the RTD and (for an accessor or mutator only) the field index from its closure, resulting in additional memory traffic. Record construction with default protocols similarly requires an indirect procedure call to the constructor procedure and a memory reference to retrieve the RTD. A constructor with programmer-supplied protocols additionally requires indirect calls to each of the protocols involved. It also requires nested procedures and rest lists to be created for each level of inheritance and additional allocation to combine the arguments into a single list to which `rc` is eventually applied.

The implementation could special case constructors for records with even more fields, and it could also special-case accessors and mutators with small field indices to avoid retrieving the computed index from the closure, but these changes are not likely to have much impact.

4. Compiler optimizations

It is often the case that the shape of the instances of a record type can be determined at compile time. In such cases, we would like to avoid as much of the procedural interface overhead as possible. In particular, record-type descriptors (RTDs) and record-constructor descriptors (RCDs) for nongenerative types should be created at compile time whenever possible, i.e., when all of the arguments to the RTD and RCD creation procedures reduce to constants. Second, record allocation should be done inline, as it is with the `vector` primitive. Third, access and mutation of record fields should be performed inline, as when constant indices are supplied to `vector-ref` and `vector-set!`. Finally, creation of record instances for nongenerative record types with only immutable fields should happen at compile time when the arguments to the construc-

```
(define find-fld
  (lambda (who rtd idx)
    (unless (record-type-descriptor? rtd)
      (error who "not a record type descriptor"
        rtd))
    (cond
      ((and (fixnum? idx) (fx>=? idx 0))
       (let ((flds (rtd-flds rtd))
             (prtd (rtd-parent rtd)))
         (let ((real-idx
                (if prtd
                    (fx+ (length (rtd-flds prtd))
                         idx)
                    idx)))
           (when (fx>=? real-idx (length flds))
             (error who
               "invalid field index for type"
               idx rtd))
           (list-ref flds real-idx))))
      (else (error who "invalid field specifier"
              idx)))))
(define record-accessor
  (lambda (rtd idx)
    (let ((offset (fld-byte (find-fld
                             'record-accessor
                             rtd idx))))
      (lambda (x)
        (unless (record? x rtd)
          (error #f "incorrect type" x rtd))
        (mem-ref x offset)))))
```

Figure 2. The `record-accessor` procedure and the `find-fld` helper

```
(define record-mutator
  (lambda (rtd idx)
    (let ((fld (find-fld 'record-mutator
                 rtd idx)))
      (unless (fld-mutable? fld)
        (error 'record-mutator
          "field is immutable" idx rtd))
      (let ((offset (fld-byte fld)))
        (lambda (x v)
          (unless (record? x rtd)
            (error #f "incorrect type" x rtd))
          (mem-set! x offset v))))))
```

Figure 3. The `record-mutator` procedure

```
(define record-predicate
  (lambda (rtd)
    (unless (record-type-descriptor? rtd)
      (error 'record-predicate
        "not a record type descriptor" rtd))
    (if (record-type-sealed? rtd)
        (lambda (x) ($sealed-record? x rtd))
        (lambda (x) (record? x rtd)))))
```

Figure 4. The `record-predicate` procedure

tor are constant, and references to a field of a constant record instance should be folded into the value of the field.

One approach would be to focus on the syntactic interface. The shape of a record type created exclusively with the syntactic interface[2] is always known at compile time. Programmers coming from languages that provide only a syntactic interface for creating records might also expect that these operations be generated statically.

This approach would likely handle well a majority of R6RS programs, but it would fail to serve programs that use the procedural interface directly, and it would not serve programmer-defined syntactic interfaces unless those are specified directly in terms of the procedural interface.

A better approach is to expand the syntactic interface into the procedural interface and use compiler optimizations to recover static record performance, taking care that the optimizations handle the expansion of the syntactic interface at least as well as direct optimization of the syntactic interface. In this way, we handle well most programs that use either interface.

With our approach, the code for record constructors, accessors, and mutators is generated at compile time when the shape of a record type can be determined. This is always the case for the expanded output of the syntactic interface and might often be the case when the procedural interface is used directly. Furthermore, constructors for nongenerative record types with only immutable fields and arguments whose values can be determined at compile time are identified and folded. Accessors for immutable fields are also folded when their arguments can be determined at compile time. Finally, RTDs and RCDs for nongenerative record types are created at compile time whenever the values of all of their fields can be determined at compile time.

4.1 Leveraging the source-level optimizer

The source-level optimizer in Chez Scheme [7] provides copy propagation, constant propagation, constant folding, and aggressive procedure inlining. The optimizer uses effort and size counters to ensure the pass runs in linear time and does not excessively expand the code. Constant folding for simple primitives is handled by calling the built-in primitives at compile time, based on a table of primitives that indicates when this is possible. For instance, the `vector-ref` primitive can be folded when the first argument is a constant vector, the second argument is a fixnum, and performing the operation does not raise an exception. More complicated primitives, i.e., those that require additional checks on their arguments, generate λ-expressions, or can only be optimized in some specific cases, are handled by a set of primitive handlers.

The optimizer works by maintaining an environment mapping known variables to one of three compile-time values. When a variable maps to another variable, copy propagation occurs. When a variable maps to a constant, constant propagation occurs. When a variable maps to a λ-expression, and the variable is referenced in call position, an inlining attempt is made. Inlining will not succeed if the effort or size counters exceed their specified bound during the inlining attempt, except that inlining always succeeds in cases where the reference is the only reference to the variable.

[2] That is, defined using `define-record-type` with a parent, if any, specified using the `parent` form and defined exclusively with the syntactic interface

4.2 Extending the source-level optimizer

We use the ability to define primitive handlers to perform compile-time optimization of the procedural interface. We also extend the optimizer's set of compile-time values to include partially static [15] representations of RTDs and RCDs when they cannot be reduced to constants, thus allowing the optimization of generative record types and of nongenerative record types for which certain pieces of information, such as opaqueness, cannot be determined at compile time. The remainder of this section describes the primitive handlers for each of the primitives that comprise the procedural interface. We start with `make-record-type-descriptor` since it is the starting point for run-time record-type creation.

The `make-record-type-descriptor` primitive handler can create the RTD at compile time when all of the arguments to the procedure are known and the record type is nongenerative. This fits well with the source inliner, but it does not allow us to optimize generative record types, since the RTD for a generative record type must be created each time the specification for the record is encountered. Even when the final RTD is not known, however, it is still possible to generate constructor, predicate, accessor, and mutator λ-expressions for the record type when the shape of the record is known at compile time.

The only two pieces of information needed to determine the shape of the RTD are the parent RTD and the number of fields specified in this record type. With these two pieces of information, it is possible to determine how much space must be allocated when creating a new record instance and the byte offsets of each field in the record at compile time[3]. In particular, information about the name of the record type, whether it is opaque, and whether it is sealed is not needed.

In such cases, the handler produces a partially static representation of the RTD. The partially static representation has two parts: a compile-time RTD and an expression that will evaluate, at run time, to the run-time RTD. The compile-time RTD is a version of the run-time RTD containing the (possibly partially static) parent RTD, if any, the field information, and any other information that happens to be available. All other information is marked unspecified.

When a fully or partially static RTD cannot be created, e.g., because the set of fields cannot be determined, the call is left unoptimized.

The value of `make-record-type-descriptor` is typically bound to a variable, at least after inlining of helpers that might otherwise obscure the binding. When the handler produces a constant or partially static RTD, the optimizer's environment maps the variable to the RTD where it can be propagated to its reference sites and thus made available to the handlers for the other record primitives.

The `make-record-constructor-descriptor` primitive handler can create an RCD at compile time when the RTD is known (and not partially static), the parent RCD is omitted or known (and not partially static), and the protocol is omitted or known (i.e., its value is determined to be a quoted procedure). Otherwise, it can create a partially static RCD when the (possibly partially static) RTD is known, the (possibly partially static) parent RCD is known or omitted, and the protocol is either omitted, a known procedure, or an unassigned variable bound to a λ-expression. If the protocol is a λ-expression, a temporary is bound to the λ-expression and the temporary is used as the protocol. (This prevents unwanted code duplication in cases where the RCD is used in multiple places.) The partially static RCD contains a compile-time representation of the RCD and the expression to generate the RCD at run time. The compile-time RCD contains a (possibly partially static) RTD, the (possibly partially static) parent RCD, and the protocol expression.

The primitive handler for `record-constructor` can generate a λ-expression for the constructor whenever the (possibly partially static) RCD is known. Generation of the λ-expression is complicated by the presence of inheritance and record protocols, reflecting the run-time complexity of the `record-constructor` procedure (Figure 1). It is worthwhile to do so, since the source optimizer can help compact the code, even eliminating the generated apply and rest arguments to the nested λ-expressions in most cases. For instance, consider the `web-color` record type from Section 2. It specifies a protocol expression and inherits from the `color` record type, which also specifies a protocol. The constructor for `web-color` must call both of these protocols to gather up the field values of the new record instance, and must return the record instance through each protocol. The generated code is as follows.

```
(web-color-protocol
  (lambda parent-args
    (lambda (hex-color)
      (apply
        (color-protocol
          (lambda (name r g b)
            ($record web-color-rtd
              name r g b hex-color)))
        parent-args))))
```

The call to `$record` is generated for every record constructor and is ultimately responsible for allocating the record instance and filling it with the RTD and field values. The outermost λ-expression is generated to supply the `pargs->new` procedure expected by the `web-color` protocol. The procedure returned by the outermost λ-expression expects the value for the `hex-color` field. This procedure then applies the `parent-args` supplied when the `web-color` protocol called `pargs->new`, so that these arguments can be passed on to the `color` protocol. The `color` protocol is then passed a procedure that expects the `name`, `r`, `g`, and `b` field values and constructs a record instance. The record instance is returned to the `color` protocol's call of `new`. The `color` protocol adds the new record to the `*colors*` database. From here it is returned through the generated λ-expression to the `web-color` protocol. Thus each protocol is called in turn to supply the field values that make up the record instance and the record instance is returned through the protocols.

With the constructor code generated, the inliner can now optimize the code. It takes advantage of the fact that the `parent-args` rest argument that is passed on to the `apply` is known to be a constant list of a given length to enable further optimization. Without this information, the inliner would not be able to eliminate the call to `apply`.

The optimized output for the constructor looks like the following[4]:

```
(define make-web-color
  (lambda (name r g b)
    (let ((str (rgb->hex-color r g b)))
      (unless (symbol? name)
        (error 'make-color
          "name must be a symbol" name))
      (unless (and (fixnum? r) (<= 0 r 255)
                   (fixnum? g) (<= 0 g 255)
```

[3] The types of the fields must also be known when Chez Scheme's extended record interface is used.

[4] Ideally, `rgb->hex-color` would be called only after the r, g, and b values have been checked, but R6RS does not specify when field values are computed, so it is not something a programmer can depend upon regardless of our implementation.

```
            (fixnum? b) (<= 0 b 255))
    (error 'make-color
      "RGB value outside range" r))
  (let ((c ($record web-color-rtd
             name r g b str)))
    (set! *colors* (cons c *colors*))
    c))))
```

The optimized, generated constructor expression incorporates both protocols and completely eliminates the nested λ-expressions, rest arguments, and calls to `apply`. Furthermore, the generated λ-expression can be inlined at constructor call sites, allowing inline allocation and potentially further simplification, although in this particular example the inliner is unlikely to do this unless some of the arguments are constant, so that the code size is reduced, or there is only one call site.

The `record-predicate` handler always generates a λ-expression that calls the extended `record?` predicate, similarly to the implementation of the `record-predicate` procedure in Section 3. Generating the λ-expression in the optimizer allows it to be inlined at its call sites or called directly, unlike with the base procedural implementation.

The `record-accessor` handler generates a λ-expression containing a record check expressed using the extended `record?` predicate and the access operation expressed using a low-level `mem-ref` operator. For instance, the record accessor for the `hex-color` field of the `web-color` record is generated as follows[5].

```
(define web-color-hex-color
  (lambda (x)
    (unless (record? x web-color-rtd)
      (assertion-violationf 'moi
        "~s is not of type ~s"
        x web-color-rtd))
    (mem-ref x 21)))
```

Because the byte offset of the field is computed at compile time, the constant offset is directly included in the `mem-ref` call. When the RTD is constant, the second argument to `record?` is the quoted RTD.

Generating the λ-expression at compile time allows it to be inlined at its call sites. When an object at the call site is known to be of the correct type, for instance when a type recovery analysis has determined it to be of the correct type, the record type check can also be eliminated and the residual code at the call site is simply a memory reference.

The `record-mutator` handler is similar to the `record-accessor` handler, with the generated λ-expression taking an additional argument representing the new value and calling `mem-set!` rather than `mem-ref`. It leaves the code unoptimized if the field is not mutable so that an appropriate exception is raised at run time.

4.3 Optimizing immutable record creation

When the λ-expression for a constructor is inlined at a call site, it is possible that the record instance built there can be created at compile time. This is possible when a constructor can be folded down into a call to `$record`[6] under the following conditions: (1) the RTD must reduce to a constant and must indicate that the record is nongenerative and that all of the fields (including those of any parent record type) are immutable, and (2) the arguments to `$record` that represent the field values must reduce to constants.

This is an optimization that cannot be performed on vectors or pairs, without further analysis, since their contents are always mutable.

When a constant record is available at compile time, either because it was present in the source, inserted by a macro, or produced by a folded constructor call, calls to record predicates and record accessors are also evaluated at compile time. A predicate call is optimized into a true or false value, depending on the result of the predicate. An accessor call is optimized into the constant value stored in the accessed field.

4.4 Handling the top-level

A record-type definition that appears at the top-level of an R6RS library or program is guaranteed to be encountered only once. Because of this, a generative record type is indistinguishable from a nongenerative record type. The source inliner can make use of this observation to treat generative record types as nongenerative when they appear at the top-level of an R6RS program or library.

Chez Scheme also supports an interactive top-level, and the optimization of generative to nongenerative types also applies to it. Variables in the interactive top-level are always mutable, however, so the values of the variables bound to the RTD and RCD cannot be propagated to their reference sites in the code that creates the other record-definition products. Instead, the syntactic interface constructs the RTD and, when possible, the RCD, during macro expansion and uses identifier macros to place the constants directly at the reference sites.

5. Analysis

The optimizations presented in the preceding section recover the performance of static record types and improve on them when record instances can be constructed at compile time. To get an idea of how important this is to program performance, we tested a small number of benchmarks that make use of record types.

The benchmarks are taken from the R6RS benchmarks [4], the Computer Language Benchmarks Game [9], and some benchmarks used to test performance in Chez Scheme in the past.

The benchmarks from the Computer Language Benchmarks Game and the GC benchmark from the R6RS benchmarks all make use of record types. The other benchmarks were converted to use record types in place of vector-based records.

In addition to the benchmarks, we measured the Chez Scheme compiler compiling itself with and without the compiler optimizations described in Section 4, to demonstrate their impact on a larger program that makes extensive use of records.

The benchmarks were run on a 2.8 GHz Intel Core i7-930 CPU with 12 GB of RAM running the Fedora 13 Linux distribution. The 32-bit version of Chez Scheme was used for benchmarking. Each benchmark was run six times and CPU time for each run was

[5] The `assertion-violationf` procedure is a Chez Scheme extension that allows for formatted error messages. The `'moi` expression is used by the compiler to choose the who argument based on the name of the containing procedure, when that can be determined.

[6] It is not always possible to fold a constructor call into a call to `$record` since some expressions, such as the `set!` in the `color` protocol, cannot be folded at compile time.

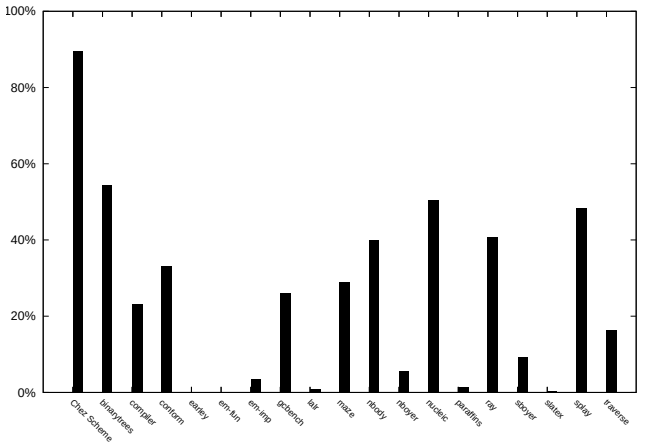

Figure 5. Percent of run-time improvement

averaged to get the time for each benchmark. On average, we saw a 21.2% improvement in run-time performance for the benchmarks, with improvements ranging from 0% to 54.4%. The percentage of improvement for each benchmark is shown in Figure 5 along with the improvement in the Chez Scheme compiler, which is a notable 89.5%. The binarytrees and nbody benchmarks come from The Computer Languages Benchmarks Game. The em-fun and em-imp benchmarks were originally written by Jefferey Mark Siskind and implement an EM clustering algorithm. The lalr benchmark performs LALR parsing. The splay benchmark is originally from CF Analysis Development and implements a splay tree benchmark. The traverse benchmark is an R6RS version of Richard Gabriel's original traverse benchmark.

We also compared the results with the vector-based versions of the benchmarks. On average, the tests were 27% slower using records with the base procedural implementation, ranging from 2.5% faster to 107% slower. On average, the tests were 4.3% faster using records with the compiler optimizations described in Section 4 enabled, ranging from 38.0% faster to 10% slower. In the cases where the vector implementation is slower than the record implementation, it is likely because the vector implementation contains an extra field to store the type of the vector, and checking to make sure it is of the correct type is often done using a combination of a vector type check, a length check, and a check of the tag, before performing the vector reference or set operation, which performs a second vector type check and range check.

Compile-time overhead for the handlers that expand calls to the record primitives into code that can be further optimized by the inliner is negligible. The code produced by the handlers gives the inliner more work to do, but the final code generated by the inliner typically contains less code, which reduces the burden on downstream passes. The net effect is that the compiler actually runs faster in most cases with the optimizations enabled than with them disabled, with overall compile times ranging from 1% slower to 21% faster in our tests.

6. Related work

Ikarus Scheme [10], Larceny [5], and PLT Racket [8], all include support for R6RS record types. All three implementations expand the syntactic interface into the procedural interface. Racket also supports a separate record system with both syntactic and procedural interfaces that is similar to the R6RS record system, but in place of protocols provides a number of options for configuring constructors with default field values and field type checks. These systems do not implement the compiler optimizations described in Section 4.

Scheme 48 [14], Chicken [18], and probably many other Scheme implementations provide a set of low-level operators for constructing a record instance from an RTD and the values of the fields, checking a record instance type, and referencing and setting record fields using an integer index to indicate the field position. Scheme 48 and Chicken (via user-committed libraries, or Eggs) support both higher level procedural and syntactic interfaces built using the low-level operators. These systems do not implement the compiler optimizations described in Section 4.

In earlier versions of Chez Scheme, the syntactic record interface expanded into a set of constructor, predicate, accessor, and mutator procedures defined in terms of a similar set of (unexposed) low-level operators. These procedures could be inlined by the source optimizer to produce code similar to that achieved with our new algorithm. Now that the procedural interface is handled well by the source optimizer, the syntactic interface more simply and cleanly expands into the procedural interface.

The idea of a procedural record system seems to be unique to Scheme, although there are several related techniques, particularly in object-oriented languages, for extending existing classes or creating entirely new classes. In dynamic languages and in languages that support either run-time evaluation of code or run-time loading of object code, techniques have also been developed to add new types at run time.

The Clojure language [12] now includes support for record-style data types through the `deftype` and `defrecord` forms. In both cases, a new type is compiled into Java byte code and loaded into the currently running Java session. While the records do not support inheritance between record types, there is an additional facility called a protocol[7] that allows several record type implementations to share a common interface. Protocols are similar to interfaces in Java, specifying a set of abstract procedures that must have concrete versions specified in the record type definition. A protocol also allows records to be used from Java. Clojure does not currently support a procedural interface for constructing record types.

The SELF language [17] is a dynamically typed object-oriented programming language. Instead of defining objects through classes, new objects are created by cloning existing objects and extending them to create new objects. This is known as protocol-based object-orient programming, and it means that similar to Scheme's procedural records, new data structures are created at run time. Instead of creating new objects as stand-alone copies, SELF uses a class-like structured called a map [3] to store information about the structure of the data type and constant values shared with other objects with the same map. Together, the objects that share a single map are referred to as a clone family. Our optimization of generative procedural record types is similar to SELF's optimization of maps, in that both commonize access to a field (or in SELF's case a slot) by using structure information common across a set of objects. The SELF optimization is attempting to deal with a more difficult problem though, in that this common structure must be divined from the actions of protocol cloning and maintained as an object is extended.

JavaScript has an object system similar to SELF, based on protocols rather than classes. Because a new property can be added to a JavaScript object at any time, objects are traditionally implemented using a dictionary data structure, like a hash table, to store the value

[7] Clojure's protocols have no relation to R6RS protocols.

of a property. The V8 JavaScript implementation [11], however, takes a different approach, using hidden classes, similar to SELF's map, to define the structure of an object. This allows a property access to be done with a memory reference to a fixed position from the start of the object, once the object's hidden class is determined. In order to ensure efficient property access, V8 compiles the call site for each property reference to a test that checks to see if the hidden class is the one expected, a jump to handler code if it is not, or a load of the property from memory. This is similar to record type field references in our system, where a test is performed to check the record type, with a jump to an error handler if it does not match, or a load form memory if it does. It is possible in Scheme to eliminate this type check when the record instance can be determined statically. Property access is more challenging to keep consistently fast, since several different classes with the same property name might all flow to the same property access point. V8 attempts to mitigate this somewhat by dynamically recompiling code when the hidden object check fails.

The Common Lisp Object System (CLOS) [6] is an object-oriented programming system for Common Lisp. It provides a `defclass` form for defining new classes and allows for multiple inheritance from existing classes. Rather than a message interface, CLOS uses generic functions that dispatch on the types of their arguments in order to provide polymorphism over these classes. CLOS is based on a meta-object protocol that can be used to alter the existing object system and created new or different object-oriented programming system. The implementation of the R6RS record system described in this paper shares the property of a meta-object protocol, or rather a meta-record protocol, by defining RTDs as records. This makes it possible to extend from the base RTD, to treat RTDs as records, and create entirely new record systems on the existing structure. One such use of this is the ftype system [13].

Mogensen [15] introduces the idea of partially static structures as a way to push partial evaluation further when part of a data structure is known statically, even if the full contents of the data structure is not. He extends an existing partial evaluator with partially static structures and states that it compares favorably with numbers reported for the original partial evaluator. The Similix partial evaluator [2] makes use of the idea of partially static structures to further partial evaluation in Scheme. The use of partially static structures in our system is limited to the RTD and RCD data types. In general, Scheme pairs, vectors, and strings, which are mutable, cannot be handled as partially static structures without further analysis to ensure that the static entries are never mutated at run time. However, the partially static structures could be extended to support record instances with one or more immutable fields, when the value of the field is known statically.

7. Conclusion

The procedural record interface to the R6RS record system provides a flexible and expressive system for defining new structured data types and a useful expansion target for the syntactic interface. This flexibility and expressivity comes at a cost. Even a carefully tuned run-time implementation of the procedural interface involves undesirable run-time definition, record construction, access, and mutation overhead. This paper shows that this overhead can be eliminated via the compiler optimizations described in Section 4 when the shape of a record type's instances can be determined at compile time.

These optimizations improve the performance of a representative set of benchmarks by over 20% on average, and Chez Scheme's compiler, which makes heavy use of records, is about 90% faster, or nearly twice as fast, with record optimizations enabled.

In our experience, generative record definitions are rarely if ever useful, and they are often confusing, especially to novice users. Our optimizer's use of partially static record-type descriptors reduces the performance hit for using generative record types significantly, but there remains at least the run-time overhead for creation of the record-type descriptor (RTD). Unfortunately, R6RS (syntactic) record definitions are generative by default. We encourage programmers to habitually include the `nongenerative` form in their record definitions, with or without an explicit UID, and we encourage the designers of future versions of Scheme to make record definitions nongenerative by default.

At present, our compiler suppresses record-type checks only when code is compiled in an unsafe mode that suppresses all type checks. It would be useful to extend the type-recovery algorithm of Adams, et al [1] to handle record types so that some checks can be eliminated even in safe code. Use of partially static structures could be extended to record instances with one or more immutable fields, allowing the known value of an immutable field to residualize to a constant even when the rest of the field values are not known.

References

[1] M. D. Adams, A. W. Keep, J. Midtgaard, M. Might, A. Chauhan, and R. K. Dybvig. Flow-sensitive type recovery in linear-log time. In *OOPSLA*, pages 483–498, 2011.

[2] A. Bondorf. *Self-Applicable Partial Evaluation*. PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen Denmark, Dec 1990.

[3] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of self a dynamically-typed object-oriented language based on prototypes. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA '89, pages 49–70, New York, NY, USA, 1989. ACM. ISBN 0-89791-333-7.

[4] W. D. Clinger. Description of benchmarks, 2008. URL `http://www.larcenists.org/benchmarksAboutR6.html`.

[5] W. D. Clinger. Larceny user's manual, 2012. URL `http://larceny.ccs.neu.edu/doc/`.

[6] L. G. DeMichiel and R. P. Gabriel. The Common Lisp Object System: An overview. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '87, pages 151–170, London, UK, UK, 1987. Springer-Verlag. ISBN 3-540-18353-1.

[7] R. K. Dybvig. *Chez Scheme Version 8 User's Guide*. Cadence Research Systems, 2009.

[8] M. Flatt, R. B. Findler, and PLT. The Racket guide, 2012. URL `http://docs.racket-lang.org/guide/index.html`.

[9] B. Fulgham. The computer language benchmarks game, 2012. URL `http://shootout.alioth.debian.org/`.

[10] A. Ghuloum. Ikarus Scheme user's guide, October 2008. URL `https://launchpadlibrarian.net/18248997/ikarus-scheme-users-guide.pdf`.

[11] Google, Inc. Chrome v8: Design elements, 2012. URL `http://developers.google.com/v8/design`.

[12] R. Hickey. Clojure, 2012. URL `http://clojure.org`.

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

[14] R. Kelsey, J. Rees, and M. Sperber. The incomplete Scheme 48 reference manual for release 1.8, 2008. URL `http://s48.org/1.8/manual/manual.html`.

[15] T. Æ.. Mogensen. Partially static structures in a self-applicable partial evaluator. In D. Bjørner, A. Ershov, and N. Jones, editors, *Partial*

Evaluation and Mixed Computation, pages 325–347. North-Holland, 1988.

[16] M. Sperber, R. K. Dybvig, M. Flatt, A. Van Straaten, R. Findler, and J. Matthews. Revised^6 report on the algorithmic language scheme. *Journal of Functional Programming*, 19(Supplement S1):1–301, 2009. doi: 10.1017/S0956796809990074. URL `http://dx.doi.org/10.1017/S0956796809990074`.

[17] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA '87, pages 227–242, New York, NY, USA, 1987. ACM. ISBN 0-89791-247-0.

[18] F. L. Winkelmann and The Chicken Team. The CHICKEN user's manual, 2012. URL `http://wiki.call-cc.org/man/4/The%20User%27s%20Manual`.