JFP **24** (6): 675–716, 2014.
doi:10.1017/S0956796814000203 First published online 1 September 2014

A run-time representation of scheme record types

ANDREW W. KEEP
University of Utah and Cisco Systems Inc.
(*e-mail:* akeep@cisco.com)

R. KENT DYBVIG
Cisco Systems Inc.
(*e-mail:* dyb@cisco.com)

Abstract

The Revised[6] Report on the Algorithmic Language Scheme added a mechanism to the Scheme programming language for creating new record types procedurally. While many programming languages support user defined, structured data types, these are usually handled syntactically, so that the compiler can make choices at compile time about the memory layout of these data types. The procedural record types in Scheme, however, can be constructed at run time, making the efficient run-time representation of record types important to ensure good run-time performance. The run-time representation used in our implementation provides an extended model for record types allowing record types to represent foreign scalar data types, e.g., machine word integers, and allows the base record type to be extended to create non-R6RS record-type systems. This article describes our run-time representation for record types, how the garbage collector handles foreign scalar data types, and includes extended record type systems both for an object-oriented programming model and a representation of foreign structured data types.

1 Introduction

The record-type system introduced by the Revised[6] Report on the Algorithmic Language Scheme (R6RS) (Sperber *et al.*, 2009) added the ability to create user-specified, aggregate data types, similar to structs or records in other languages. Record types can be created either through a procedural interface or through a syntactic interface. The two are designed to allow the syntactic interface to macro-expand into the procedural interface, with the procedural interface acting as the primitive interface to record-type creation. The record system also provides an inspection interface similar to the reflection interface in some object-oriented programming languages.

To support run-time creation and inspection of record types, each record type has an associated record-type descriptor (RTD), which is a run-time representation of the record type. The RTD can be used by the procedural interface to create:

- a predicate for identifying instances of this record type,
- accessors for accessing the fields of this record type,
- mutators for mutating the mutable fields, if any, of this record type, and
- a record constructor descriptor, which can be used in turn to create constructor procedures for the record type.

The record type system also supports inheritance, and an RTD is used to represent the parent record type when a new record type is created.

The record constructor descriptor (RCD) is created using an RTD, an optional parent RCD, and an optional *protocol procedure*. The parent RCD is used when a record type inherits from an existing record type. It allows the child RCD to utilize the parent record type's RCD protocol when specifying the parent record type's fields. The protocol in an RCD allows a programmer to specify how instances of this record type should be initialized. When a parent record type uses a protocol, the child record type is responsible only for satisfying the interface of the parent record type's protocol. Thus, the child need not be aware of changes to the parent record type, as long as the interface remains the same. This is similar to a constructor in an object-oriented programming language but supports functional rather than imperative field initialization. The protocol allows fields to be calculated from the input to the protocol or for default values to be specified. Within a protocol, the record instance being constructed can be modified or otherwise manipulated before it is returned, allowing, for example, the construction of cyclic structures or the storing of instances in a database.

Both the RTDs and the RCDs are data structures that must be represented at run time. In our implementation, both are represented as records, with an RTD record type to represent RTD records and an RCD record type to represent RCD records. A special base RTD represents the bottom of the RTD inheritance hierarchy and all record types are built from the base RTD. This provides a record meta-layer, similar to the object meta-layer in the Common Lisp Object System (DeMichiel & Gabriel, 1987). The R6RS record system is just one possible record system built upon this meta-layer. We have used this flexibility to create both an object-oriented programming system and a flexible system for accessing structured foreign-types, called ftypes.

Our implementation extends the R6RS record system to allow record types to contain foreign scalar data types, such as raw integers, floating-point numbers, and other foreign scalar data types. Since these types are represented natively (based on the application binary interface of the machine) each RTD must contain enough information to inform the garbage collector which fields are Scheme object types and which are foreign data types. This feature is used by the ftype system in order to represent pointers into foreign memory that the garbage collector is not responsible for maintaining.

This article discusses the run-time representation of record types in Chez Scheme (Dybvig, 2009) with a description of an efficient implementation of the procedural interface for the record system. It also presents the interaction of the record system with the garbage collector, including how foreign scalar fields are handled by the collector. Finally, we present the object-oriented programming system and the ftype system and describe how these systems are built using the record system meta-layer.

The remainder of this article is organized as follows. Section 2 reviews the R6RS record system. Section 3 describes Chez Scheme specific extensions to the R6RS record system. Section 4 describes our representation of R6RS record types and how the garbage collector handles records with our extensions. Section 7 describes our procedural implementation of the record system. Section 8 provides an overview of compiler optimizations used to improve performance of procedural record types. Section 9 describes the object system

and foreign-type interface built on top of the record system. Sections 10 and 11 discuss related work and conclusions.

2 R6RS record types

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 `color` record type that contains `r`, `g`, and `b` fields, using the procedural interface.

```
(define color-rtd
  (make-record-type-descriptor 'color #f #f
    #f #f '#((mutable r) (mutable g) (mutable b))))
(define color-rcd (make-record-constructor-descriptor color-rtd #f #f))
(define make-color (record-constructor color-rcd))
(define color? (record-predicate color-rtd))
(define color-r (record-accessor color-rtd 0))
(define color-r-set! (record-mutator color-rtd 0))
(define color-g (record-accessor color-rtd 1))
(define color-g-set! (record-mutator color-rtd 1))
(define color-b (record-accessor color-rtd 2))
(define color-b-set! (record-mutator color-rtd 2))
```

The `color` record-type descriptor (RTD) specified here does not inherit from any other RTD, indicated by the first `#f`. It is generative, so `make-record-type-descriptor` creates a new (and unique) RTD each time it is called, indicated by the second `#f`. 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, indicated by the third `#f`. The RTD is not opaque, meaning the RTD can be extracted from an instance of the record type and used to inspect or even modify its fields, indicated by the fourth `#f`. Finally, the RTD specifies three mutable fields `r`, `g`, and `b`.

Creating the constructor for the `color` 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 `color` record type is generated with the RCD. The predicate, accessors, and mutators for the `color` record type are generated with the RTD and, in the case of the accessors and mutators, the index of the field. The index is zero based, starting from the first field of this record type. If the record type inherits from a parent record type, the index is still zero based, and accessing the fields of a instance of this record contributed by the parent RTD is done using the accessors from the parent. Because fields are identified by indices, two or more fields can have the same name.

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

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

https://doi.org/10.1017/S0956796814000203 Published online by Cambridge University Press

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 `color-rtd` and `color-rcd` variables are not directly visible via the syntactic interface, though the syntactic interface does provide syntactic forms for obtaining the RTD and RCD from a record name. In addition to the `fields` clause, the `define-record-type` form 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.

To avoid duplicate bindings for accessors and mutators, the fields of a record specified via the syntactic interface must be distinctly named, though a child's field names need not be distinct from its parent's field names.

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 one reason 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 child protocol 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 backward compatible.

A protocol is analogous to a constructor or initialization 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 an immutable color record that also stores a name. We could create a variation on the color record described above, 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 fixnums between 0 and 255. These restrictions could be enforced through

https://doi.org/10.1017/S0956796814000203 Published online by Cambridge University Press

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) (fixnum? g) (fixnum? b))
          (error 'make-color "RGB values must be fixnums" r g b))
        (unless (and (<= 0 r 255) (<= 0 g 255) (<= 0 b 255))
          (error 'make-color "RGB value outside range" r g b))
        (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 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 works without any changes. However, since the record type is generative, if it was moved into a context where it could be executed more than 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.[1]

```
(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) (fixnum? g) (fixnum? b))
          (error 'make-color "RGB values must be fixnums" r g b))
        (unless (and (<= 0 r 255) (<= 0 g 255) (<= 0 b 255))
          (error 'make-color "RGB value outside range" r g b))
        (let ([c (new name r g b)])
          (set! *colors* (cons c *colors*))
          c)))))
```

[1] Generative record definitions are rarely required, sometimes lead to confusion with type mismatches on seemingly identical types, and generally have more run-time overhead. Despite this, the language designers chose to make generativity the default. Programmers should therefore develop the habit of including a nongenerative clause in every record definition.

https://doi.org/10.1017/S0956796814000203 Published online by Cambridge University Press

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 `pargs->new` that expects the parent-protocol arguments. This procedure returns another procedure that expects the child-field values (in this case, just the value of `hex-color`) and returns a new instance of the record type with the parent fields initialized according to the parent protocol and the child fields initialized to the supplied child field values. When a new `web-color` is created, it should be added to the global `*colors*` list, just as a new `color` record is added. Since the protocol for `color` is invoked as part of the process of creating the new `web-color`, the `web-color` protocol not need to do so directly.

The `web-color` protocol requires the user of the `web-color` record type to supply the color twice, once as a web color and once as a set of RGB values. This might lead to inconsistencies, since the web color string and the RGB values might specify different colors. The protocol could verify consistency, but a better option is to generate one representation from the other and save the programmer from the inconvenience of specifying the color twice. We can do so as follows 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 `r`, `g`, and `b` values to the hexadecimal string representation. This value is then filled in for the `hex-color` field of the `web-color` record type. We could just as easily have the protocol accept a hexadecimal string and perform the reverse conversion, or even use a `case-lambda` form (Sperber *et al.*, 2009) to allow either `r`, `g`, and `b` values or a hexadecimal string.

https://doi.org/10.1017/S0956796814000203 Published online by Cambridge University Press

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` and `r` values:

```
(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"
(color-r red)  ⇒  255
```

2.2 The inspection interface

It is 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 0)            ⇒  #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 Extensions to R6RS record types

Our implementation of the R6RS record system supports an alternative interface that predates the R6RS. In addition to having a slightly different model for the procedural and syntactic interfaces from the R6RS record system, the alternative interface supports records with foreign scalar data type fields. This allows a record to store raw integers, un-tagged floating point numbers, and raw machine pointers, in addition to Scheme data types. Record types that make use of these features cannot be built using the standard R6RS interface, and instead rely on the alternative interface. Internally, there is no difference between record types created through the R6RS interface and those created using the alternative interface, and it is possible for record types created using the alternative interface to inherit from R6RS records and vice versa. There is, however, one small difference between the two: because the R6RS syntax does not support foreign scalar data types, our system does not support creating an RCD for a record that includes foreign scalar data types. If we were to extend the R6RS syntax to support this, it should be possible to also extend the RCD and

https://doi.org/10.1017/S0956796814000203 Published online by Cambridge University Press

protocols implementation to support it. The RCD is not supported because the protocol-handling code does not provide the functionality to check and convert Scheme data into foreign scalar data. A constructor procedure can still be created for these record types using the older record system's procedural or syntactic interface.

Record types created using the alternative interface are never opaque or sealed. Record types created using the alternative syntactic record interface are always nongenerative, unlike those created through the R6RS interface, which can be generative or nongenerative.

With the alternative procedural interface, `make-record-type` is used to create an RTD. Since RCDs are not supported by the alternative interface, record constructors are created directly from the RTD and always accept one argument for each field. The example below creates a record constructor with a field that has a default value by wrapping the record-constructor procedure with a procedure that supplies the default value.

```
(define EX1-rtd
  (make-record-type "EX1"
    '((mutable float a) (mutable integer-32 b)
      (mutable c) (mutable d)))))
(define make-EX1
  (let ([rcons (record-constructor EX1-rtd)])
    (lambda (a b c)
      (rcons a b c #f))))
(define EX1? (record-predicate EX1-rtd))
(define EX1-a (record-accessor EX1-rtd 0))
(define EX1-b (record-accessor EX1-rtd 1))
(define EX1-c (record-accessor EX1-rtd 2))
(define EX1-d (record-accessor EX1-rtd 3))
(define set-EX1-a! (record-mutator EX1-rtd 0))
(define set-EX1-b! (record-mutator EX1-rtd 1))
(define set-EX1-c! (record-mutator EX1-rtd 2))
(define set-EX1-d! (record-mutator EX1-rtd 3))
```

This creates a new record type, `EX1`, with four fields:

- `a`, a mutable field that contains a float foreign scalar,
- `b`, a mutable field that contains an integer-32 foreign scalar,
- `c`, a mutable field that contains a Scheme object, and
- `d`, a mutable field that contains a Scheme object.

The procedure wrapping the constructor sets `d` to the default value `#f`.

The syntactic interface is invoked through the `define-record` form. The `EX1` record type can also be created using the syntactic interface, including creating a constructor with the default value for the `d` field.

```
(define-record EX1
  ((mutable float a) (mutable integer-32 b) (mutable c))
  (((mutable d) #f)))
```

The `define-record` form creates

- a constructor procedure, `make-EX1` in this case;
- a predicate, `EX1?` in this case;
- an accessor for each field, `EX1-a`, `EX1-b`, `EX1-c`, and `EX1-d` in this case; and

https://doi.org/10.1017/S0956796814000203 Published online by Cambridge University Press

- a mutator for each mutable field, `set-EX1-a!`, `set-EX1-b!`, `set-EX1-c!`, and `set-EX1-d!` in this case.

It also creates a compile-time binding for the identifier `EX1`, which allows the RTD for this record type to be retrieved with `record-type-descriptor` or used as a parent in other `define-record` or `define-record-type` forms.

When a record type contains foreign scalar fields, as `EX1` does, the generated record-constructor procedure verifies that the argument supplied for each foreign-type field is valid for that foreign type and converts the value to the appropriate foreign representation before storing the value in the field. Likewise, the mutator created for a foreign-type field verifies that the supplied value is valid for the foreign type and converts it to the foreign representation before storing it. An accessor for a foreign-type field is responsible for performing the opposite conversion.

Our newly created `EX1` record type can be used as follows:

```
(define ex1 (make-EX1 5 100 "bob"))  ⇒
Exception: invalid value 5 for foreign type float
(define ex1 (make-EX1 4.75 100 "bob"))
(EX1? ex1)                           ⇒  #t
(EX1-a ex1)                          ⇒  4.75
(EX1-b ex1)                          ⇒  100
(EX1-c ex1)                          ⇒  "bob"
(EX1-d ex1)                          ⇒  #f
(set-EX1-b! ex1 (expt 2 33))         ⇒
Exception: invalid value 8589934592 for foreign type integer-32
(EX1-b ex1)                          ⇒  100
(set-EX1-b! ex1 1024)
(EX1-b ex1)                          ⇒  1024
```

The checks for the `a` field in the `make-EX1` constructor prevent us from using `5` as the value for a floating point field (though `5.0` would have worked). Similarly, if we try to set the value of the `b` field to 2^{33}, the check in the `set-EX1-b!` mutator recognizes that this value is outside the valid range for this field and raises an exception to this effect, preventing the field value from being changed.

4 Record type representation

For each new record type, a new record type descriptor (RTD) is created. The RTD for a record type is the central repository of information about that record type. It is used by the procedural interface to create a predicate procedure, field accessors, and field mutators for the record type. It is also used by the inspection interface to retrieve information about the record, including field names, information about fields, and the parent record type (represented as an RTD). Internally, the RTD is used as a type tag to differentiate a given record from built-in Scheme data types and other records that are not in the inheritance hierarchy of the record's RTD. It is also used by the garbage collector to determine the size of record instances, which fields contain Scheme data types, and which of those are mutable.

Each record instance has a slot for each field defined by the record type (and its parent, if the record type inherits from another record type) and an extra slot to store its RTD. If

https://doi.org/10.1017/S0956796814000203 Published online by Cambridge University Press

the record type contains foreign data type fields, adjacent fields are packed into the same slot when they are smaller than a single machine-sized word. If the foreign data type fields are not adjacent, they are each padded to occupy a full machine-sized word slot to provide proper alignment for the Scheme object fields. A foreign field may also take up more than one slot, if it is larger than the machine-sized word. The Scheme object fields, however, are always word-aligned. The slot for the RTD is used to determine the type of the record instance, which differentiates it from built-in Scheme types and from other record types. The RTD stored in the record can also be retrieved through the inspection interface to allow a program, such as a debugger or meta-circular interpreter, to determine information about the record.

RTDs are themselves represented using records. This means that each RTD is an instance of the RTD record type and has a slot for the RTD record type. This sets up a hierarchy that is rooted at the base RTD. The base RTD has itself in its RTD slot.

RTDs have eight fields: parent, size, Scheme object field (*pointer*) mask (`pm`), mutable Scheme object field (pointer) mask (`mpm`), name, fields, flags, and unique identifier (UID). The *parent* field stores the parent RTD, if there is one, and `#f`, if there is not. The *size* field indicates the size of an instance of this record type in bytes. The *Scheme object mask (`pm`)* field is a bit mask that indicates which slots hold Scheme object fields and which hold one or more scalar foreign fields. The *mutable Scheme object field mask (`mpm`)* is a bit mask that indicates which Scheme object fields are mutable. The *name* field records the name specified for the field. The *fields* field contains a list of entries each describing a field of this record type; each entry contains the name of the field, a flag to indicate if the field is mutable, the type of the field, and a precomputed byte-offset into the record instance where the field can be found. The mutable indicator in the *fields* field and the mutable bits in the Scheme object mutable pointer mask (`mpm`) might differ, since mutable foreign scalar data fields are marked as mutable in the *fields* list, but not in in the Scheme object mutable pointer mask, since while the field is mutable, it is not a Scheme object. The *flags* field records whether the record type is generative, whether the record type is opaque, and whether the record type is sealed. Finally, the *UID* field records the UID for this record type. For record types where no UID was provided, i.e., generative records, a new globally unique symbol is created when the record-type descriptor is constructed.

The base RTD has no parent, and all of its fields are immutable Scheme objects. The name and UID are both the Scheme symbol `$base-rtd`. It is nongenerative, not opaque, and not sealed. The field entries for the base RTD describes its own fields and those that must be present in any other RTD, i.e., the RTD fields described above. Figure 1 shows the layout of the base RTD.

When a new record type is created, a new RTD, i.e., an instance of the RTD type, is constructed. For example, an RTD for the `color` record type from Section 2 can be seen in Figure 2. The RTD slot points to the base RTD, since the color RTD is an instance of the RTD record type. The parent field is `#f` since the color record type does not inherit from any other record type, and the fields list contains entries for the `name`, `r`, `g`, and `b` fields, which are all immutable Scheme object fields. The flags value, 0, reflects the fact that the color record type is nongenerative, not opaque, and not sealed. The UID is a generated symbol based on the name (“color”); this symbol is generated once and for all during macro expansion, since the record type is nongenerative.

https://doi.org/10.1017/S0956796814000203 Published online by Cambridge University Press

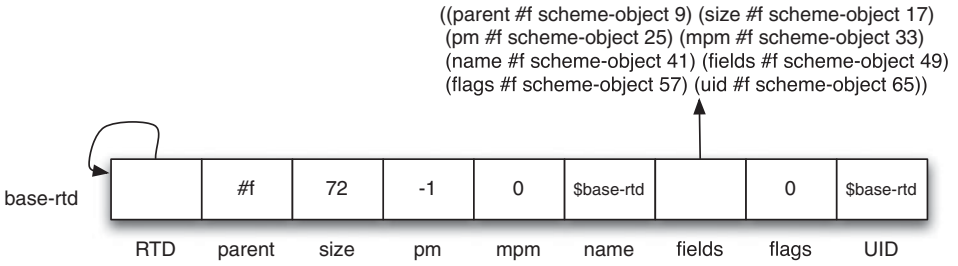

Fig. 1. The base RTD.

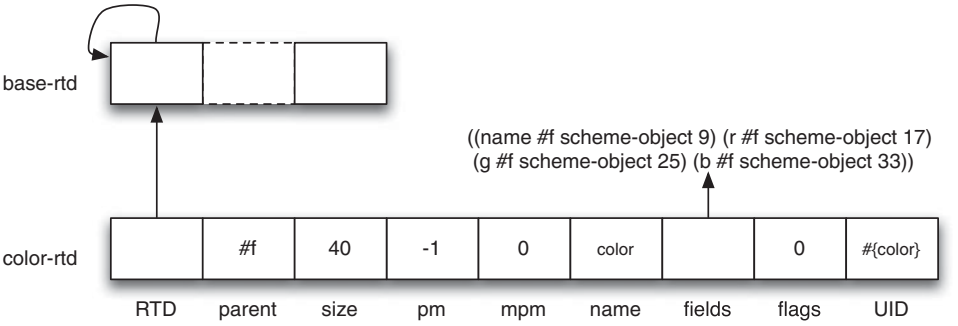

Fig. 2. The color RTD.

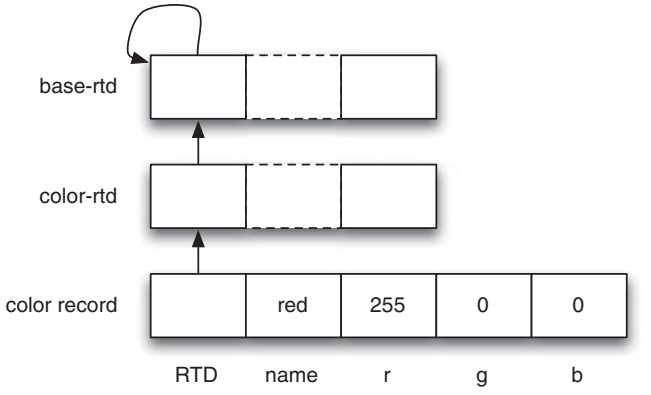

Fig. 3. A color record.

The structure of a `color` record can be seen in Figure 3. As expected, the structure of the `color` record is similar to the structure of the RTDs that we have already seen, since RTDs are records. The color record has five total slots, one for the RTD pointer and one for each of the four fields: `name`, `r`, `g`, and `b`. The color record in the example is the color record for red, where the name is the Scheme symbol `red`, the `r` field contains `255`, indicating maximum red saturation, and the remaining fields are `0` indicating no saturation of green or blue.

Figure 4 shows the RTD for the `web-color` record type. The `web-color` record type inherits from the `color` record type, so the parent field has a pointer to the `color` RTD. The RTD slot for the `web-color` record type is still the base RTD. The fields field of

https://doi.org/10.1017/S0956796814000203 Published online by Cambridge University Press

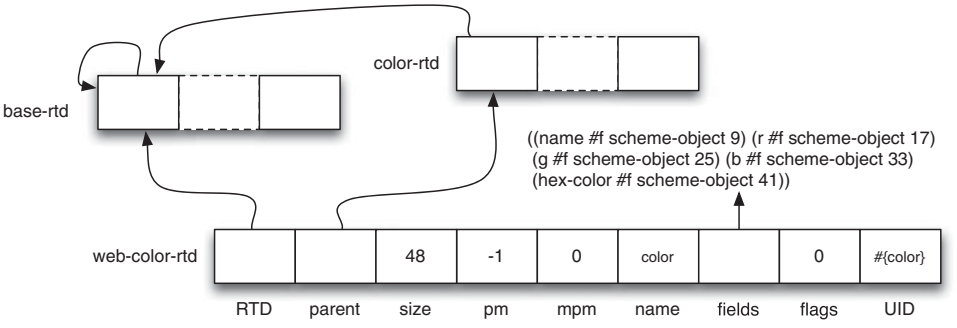

Fig. 4. The web-color RTD.

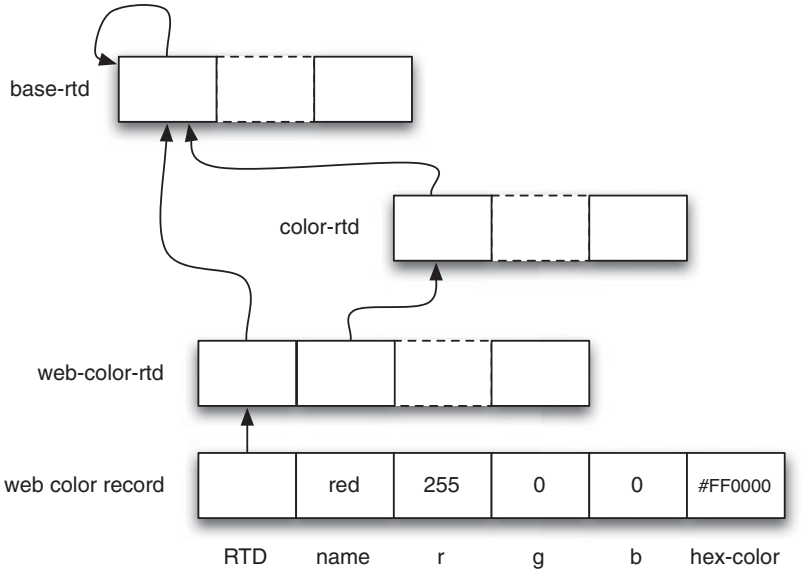

Fig. 5. A web-color record.

the `web-color` record contains the precalculated locations the full set of fields, both those from the `color` record type and the `web-color` record type. This was a somewhat arbitrary design decision to simplify various operations; the fields field could instead hold only entries for the child fields.

An example `web-color` record can be seen in Figure 5. This record has a slot for the RTD, a slot for each of the four fields of its parent, and a slot for its single additional field, the `hex-color` field. The record hierarchy is entirely captured by the structure of the RTD, so there is no need for any additional slots in the record representation. Since the record system supports only single inheritance, fields inherited from a parent RTD share the same memory layout as the records of the parent RTD. This allows accessors generated for the parent to access fields of the child type, without translating the field offsets. The example `web-color` record is also for the color red, with the additional `hex-color` field filled with the string `#FF0000`.

https://doi.org/10.1017/S0956796814000203 Published online by Cambridge University Press

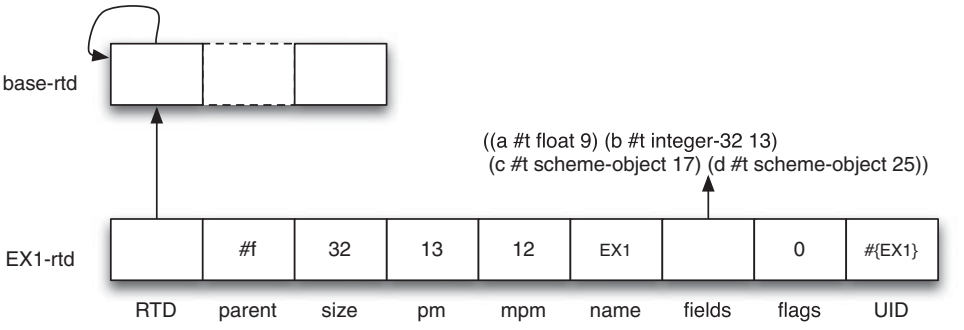

Fig. 6. The EX1 RTD.

In these examples, all of the fields have been immutable Scheme object fields, so the Scheme object mask (`pm`), mutable Scheme object field mask (`mpm`), mutability flag, and field type are the same in all record types and fields. Figure 6 shows the example from Section 3. The first difference between the `EX1` RTD and the other RTDs we have looked at is the Scheme object mask. In the case of the `EX1` RTD the mask is `13` or `1101` in binary. The bit mask can be read from right to left, with the right most `1` corresponding to the first machine-word-sized slot, which holds the RTD. The bit is set (1) to indicate the machine-word sized slot contains a Scheme object and unset (0) to indicate it contains foreign scalar data. Since the RTD is a Scheme object, it is marked with a `1`. The second machine-word-sized slot holds both the `float` field `a` and the `integer-32` field `b`. This is because these numbers were generated on the 64-bit version of our system, where the `float` and `integer-32` can be packed into the same 64-bit word. The next two `1`s correspond to the fields `c` and `d` which are both Scheme object fields.

The mutable Scheme object field mask is also different from our first two example RTDs. The value is `12` or `1100` in binary. Here, the first slot, which contains the RTD, is marked as `0` because it is immutable. The second slot, which contains the `a` and `b` fields is also marked as `0`, because while it contains two mutable fields, these fields contain foreign data. The third and forth slots, which contain the `c` and `d` fields, are both marked with `1`, because they contain mutable Scheme objects. The Scheme object field mask and mutable Scheme object field mask are used by the garbage collector, as we discuss in Section 5.

Figure 7 shows an example of an `EX1` record. The record has four machine-word-sized slots, one for the RTD, one shared by the `a` and `b` fields, one for the `c` field, and one for the `d` field. Here, the `a` field contains the raw floating point number `4.75`, the `b` field contains the raw 32-bit integer number `100`, the `c` field contains a pointer to the Scheme string `"bob"`, and the `d` field contains the Scheme object `#f`. Scheme object fields are always aligned in the machine-word-sized slots, because they are always represented by a tagged machine-word pointer.

5 Garbage collection of record types

A discussion of the full storage management system is beyond the scope of this article. It is useful to understand the basics, however, in order to better illustrate how records are handled within the garbage collector. Our system uses a generational, copying garbage

https://doi.org/10.1017/S0956796814000203 Published online by Cambridge University Press

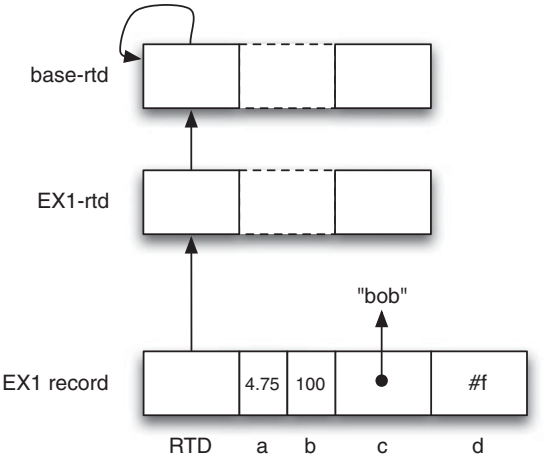

Fig. 7. An EX1 record.

collector. New objects are allocated in a single, per thread, allocation area,[2] which allows efficient inline, lock-free allocation. The garbage collector is triggered asynchronously once a user-definable amount of allocation has occurred. When the garbage collector runs, objects in the common allocation area are separated into different heap segments based on the object type or properties of the object type (Dybvig *et al.*, 1994). A *dirty vector* is used to track cross generational links that lead from an older generation to a newer generation. These "reverse" generational links can exist only in Scheme objects with mutable fields, such as pairs, vectors, and records with mutable Scheme object fields.

When collecting a record, the garbage collector uses the Scheme object field mask (`pm`) to determine which fields contain Scheme objects, and hence which contain pointers that must be traced during the garbage collection process. The Scheme object mutable field mask (`mpm`) determines which fields that contain Scheme objects are mutable, and hence which could be added to the dirty vector. The garbage collector also uses these two bit masks to determine the segment a record should be copied to.

When the garbage collector determines that a record is live, it is copied into one of four heap areas, based on the field types and mutability.

Any record can be copied to the *impure record* area. The collector expects objects in this area to be record objects that might contain a combination of Scheme object and scalar foreign data fields, with Scheme object fields that might be mutable. Since the area generally combines Scheme object pointers and scalar foreign data, the collector must walk each bit in the Scheme object field mask to determine which "slots" contain Scheme fields, and should be swept as part of the collection. It must also have a dirty vector to retain reverse-generational links. This makes this a general, but somewhat inefficient, area in which to store record objects.

In order to mitigate this inefficiency, certain record objects are placed in one of three other areas, depending on the properties of the record type. When a record object contains

[2] In the single-threaded version there is a single new-object allocation area; in the multi-threaded version, each language-level thread corresponds to an operating-system thread with its own allocation area.

https://doi.org/10.1017/S0956796814000203 Published online by Cambridge University Press

a mix of Scheme object and foreign data fields, and the Scheme object fields are all immutable, the record object is stored in the *pure record* area. The collector must still use the Scheme object field mask to determine which "slots" should be swept, but it no longer needs a dirty vector, since the area cannot contain reverse generational links.

When a record contains only Scheme object fields and one or more of those fields is mutable, the record object is put in the *impure* area. Since all of the fields are Scheme objects, the collector can simply sweep all of the pointers in this area. Other Scheme objects that contain only Scheme objects, such as pairs, can also be stored in this area. This area requires a dirty vector, since it can contain reverse generational links.

When a record contains only Scheme object fields and all of the fields are immutable, it is stored in the *pure* area. This is the most efficient place to store records, since all of the "slots" are known to be Scheme objects and, hence, can be swept without checking the Scheme object field mask. There is also no need for a dirty vector, since the area cannot contain reverse generational links.

The garbage collector uses the Scheme object field mask (`pm`) to determine whether a slot contains Scheme object data. When a field contains Scheme object data, the field is treated as any other Scheme object, and if the Scheme object is a pointer, i.e., not an immediate value, it must be followed during the garbage collection process, since it is still live and needs to be copied into the new generation, and the pointer updated to point to the new location of this Scheme object. When a slot in the record is marked as a foreign data type, the value is simply copied to the new location of the record. The mask serves an important purpose, because without it a slot in the record might be mis-identified as a Scheme object, causing the garbage collector to attempt to follow an invalid Scheme object pointer.

When older-generation impure-record areas are marked in the dirty vector as possibly containing reverse generational pointers and scanned by the collector, the collector uses the mutable Scheme object field mask (`mpm`) in each scanned record object's RTD to determine which fields to trace.

6 Reading and writing records

Like most built-in Scheme objects, records and RTDs can be written via the Scheme printer and read via the Scheme reader. Records written by the printer can be read by the reader only when the RTD for the record is loaded and attached to the same UID that was used when the record was written. This is because a record instance written by the printer includes only the record fields and the UID of the record type and not the RTD itself. Our system also provides a mechanism that allows Scheme records to be written with a programmer-defined writer procedure.[3] The default record writer prevents an RTD written using the printer from being read with the reader to avoid potential problems of reading machine specific information into a different version of Scheme or a Scheme running on a different machine type, which might require different machine specific information.

Our system also provides a method for writing Scheme data in binary form through the *fast loading* (FASL) format. Records that are written through the FASL format can always

[3] It is often helpful to define a record writer to hide some fields or to format the record in some special way, even though the record can no longer be recreated by the Scheme reader.

https://doi.org/10.1017/S0956796814000203 Published online by Cambridge University Press

be read in using the FASL reader, because the RTD is also written out to the file. The FASL format contains information about the version and machine type of the Scheme that created the file, which is what enables us to write the machine-specific information about an RTD to the file. If more than one record instance of the same record type is written to the file, the RTD is written to the file only once—just like any other object with multiple occurrences. When a record is loaded through the FASL system, and its RTD is already loaded in the current Scheme system, it is checked for compatibility but not duplicated. The UID attached to each RTD makes this possible and only nongenerative records have the potential to have conflicting definitions attached to the same name. If a nongenerative record has changed between Scheme sessions and a data file with an older format is loaded, the compatibility check causes an exception to be raised.

7 An efficient procedural implementation

The layout of RTDs and records is just one part of building an efficient run time for the record system. Record constructor, predicate, accessor, and mutator procedures must also be written to ensure efficiency. When an RTD is constructed, 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. This information is used to fill in the fields of the RTD discussed in the previous section.

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 and that the fields of the RTD are all Scheme object fields. If the record type contains foreign data type fields, an RCD cannot be created, but a constructor can be made directly with an extended version of the `record-constructor` procedure that operates on either an RTD or an RCD. 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 8.

https://doi.org/10.1017/S0956796814000203 Published online by Cambridge University Press

```
(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)] [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))))
```

Fig. 8. The `record-constructor` procedure.

https://doi.org/10.1017/S0956796814000203 Published online by Cambridge University Press

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 calls the internal record constructor `$record` with the RTD and the values of the fields. If the number is larger than six, it defaults to using a rest argument and calls `$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 hierarchy 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.

When a record type has foreign data type fields, an RCD cannot be created for the record type. Instead, the record constructor procedure is generated directly from the RTD. This is similar to the internal `rc` procedure in Figure 8, since without the RCDs, there are no protocols. However, any foreign data type field needs to be checked to ensure that the constructor argument is a Scheme object that can be converted into the appropriate type and then the conversion must take place. This is handled by checking each argument that fills a foreign field and performing the conversion before the field value is set.

The `record-accessor` procedure takes an RTD and an index into the field list constructed by `make-record-type-descriptor`. It uses the `find-fld` helper to find the field offset as shown in Figure 9. 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.

Like the `record-accessor` procedure, the `record-mutator` procedure requires information about the field. 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 10.

The `record-accessor` and `record-mutator` for a foreign data type field are slightly more complicated than those for Scheme object fields. The accessor must convert the foreign data representation into a Scheme object representation. The mutator must check that the new value is a valid Scheme object for the foreign field type and convert the value to the foreign representation. The conversions are managed through the foreign reference and foreign set procedures (Dybvig, 2009).

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

https://doi.org/10.1017/S0956796814000203 Published online by Cambridge University Press

```
(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)))))
```

Fig. 9. 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))))))
```

Fig. 10. 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)))))
```

Fig. 11. The `record-predicate` procedure.

is shown in Figure 11. The `$sealed-record?` predicate is potentially faster, 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 is about as efficient as we can make it without further compiler optimizations. 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.

https://doi.org/10.1017/S0956796814000203 Published online by Cambridge University Press

8 Compile-time optimization of procedural records

A constructor, predicate, accessor, or mutator created 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.

While the procedural implementation is efficient, generating record constructors, predicates, accessors, and mutators at run time inhibits other compile-time optimizations, like inlining, that can improve the performance considerably. In particular, we would like the record system to provide performance on par with the performance of syntactic aggregate data types, such as C's structs.

One approach to these optimizations would be to focus on the syntactic interface and optimize those record definitions that have a fully syntactic definition. The shape of a record created entirely with the syntactic interface[4] is always known at compile time.

This approach would likely handle the majority of R6RS programs, but it would provide no optimizations for programs that directly use the procedural interface. This also would not help situations where programmers decide to define their own syntactic interfaces that expand into the procedural interface.

A better approach is to expand the syntactic interface into the procedural interface and use compiler optimizations to optimize the procedural interface. To this end, we have extended our existing source optimizer (Waddell & Dybvig, 1997) to support optimizing the procedural interface described in this section. The source optimizer performs 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 `list-ref` primitive can be folded when the first argument is a constant list, the second argument is a constant fixnum, and performing the operation does not raise an exception. More complex primitives, i.e., those that require additional checks on their arguments, generate λ-expressions, or can be optimized only in some specific cases, are handled by a set of primitive handlers.

[4] That is, defined using `define-reocrd-type` with a parent, if any, specified using the `parent` form and also defined entirely using the syntactic interface, or a record type defined using the `define-record` syntactic form.

https://doi.org/10.1017/S0956796814000203 Published online by Cambridge University Press

When the shape of a record is known statically, either because it was created through the syntactic interface, or because it was created using a constant fields vector and constant parent RTD, the record type can be optimized such that:

- record accessors can be reduced to a type check and a single memory reference,
- record mutators can be reduced to a type check and a single memory set,
- record predicates can be reduced to an inline type check, and
- when the record protocol is simple enough, allocation for creating a new record instance can be performed inline.[5]

These optimizations allow the Scheme procedural record system to perform on par with more traditional, purely syntactic aggregate data structures, when the size and layout of the data structure is known statically. We tested these optimizations over our existing efficient interface, and found a 21.2% improvement in run-time performance on a set of benchmarks, with improvements ranging from 0% to 54.4%, largely depending on how heavily the benchmark made use of the record system. More information about the record optimizations can be found in our Scheme Workshop paper (Keep & Dybvig, 2012).

9 Extending the base RTD

Using records to represent RTDs provides a simple record meta-layer that can be extended to allow different record systems to be defined, similar to the object-meta-layer in the Common Lisp Object System (DeMichiel & Gabriel, 1987). The R6RS record system (and our extensions to it) create new record types by creating a new RTD that has the base RTD as its RTD field and, when it inherits from an existing record, another RTD as its parent. Since RTDs are themselves records, we can extend the base RTD to create a new type of RTD. As with an extension of any other record, the new RTD type has all the fields of an RTD, with any additional fields needed for the new RTD type. When we create new RTDs of the new record type, their RTD field is filled with the extended base RTD in place of the base RTD.

We have used this flexibility to create both an object-oriented programming system (Waddell & Dybvig, 2004) and a system for representing structured foreign types (Keep & Dybvig, 2011). The next two sections describe how we extend from the base RTD to create these two systems.

9.1 An object system based on record types

Before describing the implementation of our object-oriented programming (OOP) system, it is helpful to understand the basic features it provides. The OOP system is similar to many OOP languages, such as Java or Ruby, that support single inheritance classes with a base class, in our case named `<root>`. Similar to Java, our system also supports interfaces to allow classes that do not share the same inheritance hierarchy to still implement methods from the same interface. Interfaces define the set of methods that must be implemented

[5] Record access and mutation can be further reduced to a single memory reference or a single memory set, if the record type of the object is known statically.

https://doi.org/10.1017/S0956796814000203 Published online by Cambridge University Press

when a class implements this interface. Interfaces can also inherit, using single inheritance, from existing interfaces. A class can inherit only from another class and an interface can inherit only from another interface.

Each class definition creates a constructor for the class, a predicate for the class, a set of accessors for the public instance variables of the class, a set of mutators for the public mutable instance variables of the class, and a set of user defined methods. For instance, we can create a class to store two-dimensional points as follows:

```
(define-class (<point-2D> x y) (<root>)
  (ivars [public x x] [public y y])
  (methods
    [point=? (o) (and (= x (<point-2D>-x o)) (= y (<point-2D>-y o)))]
    [point+ (o)
     (make-<point-2D> (+ x (<point-2D>-x o)) (+ y (<point-2D>-y o)))]
    [point- (o)
     (make-<point-2D> (- x (<point-2D>-x o)) (- y (<point-2D>-y o)))]))
```

The `define-class` form defines a new class named `<point-2D>`. The constructor expects two arguments, `x` and `y`, specified immediately after the name: `(<point-2D> x y)`. The `<point-2D>` class does not inherit from any other class, so it specifies `<root>` as its base class.[6] The class specifies two public instance variables `x` and `y` in the `ivars` clause that take their values from the `x` and `y` values supplied to the constructor in the `ivars` clause. The class also specifies the `point=?` method for comparing this point to another point, the `point+` method for adding this point to another point, and the `point-` method for subtracting another point from this point. All three methods expect a single additional argument representing the other point involved in the operation.

We can use our newly defined class to perform calculations on our points:

```
(define pt1 (make-<point-2D> 1 1))
pt1                                  ⇒  #<instance of <point-2D>>
(<point-2D>? pt1)                    ⇒  #t
(<point-2D>-x pt1)                   ⇒  1
(<point-2D>-y pt1)                   ⇒  1
(define pt2 (make-<point-2D> 13 5))
(define pt3 (make-<point-2D> 1 1))
(point=? pt1 pt2)                    ⇒  #f
(point=? pt1 pt3)                    ⇒  #t
(point=? pt3 pt1)                    ⇒  #t
(define pt4 (point+ pt1 pt2))
(<point-2D>-x pt4)                   ⇒  14
(<point-2D>-y pt4)                   ⇒  6
(define pt5 (point- pt2 pt3))
(<point-2D>-x pt5)                   ⇒  12
(<point-2D>-y pt5)                   ⇒  4
```

New instances of the `<point-2D>` class can be created with `make-<point-2D>`. The `<point-2D>?` predicate tests to see if a Scheme object is an instance of the `<point-2D>` class. The `<point-2D>-x` and `<point-2D>-y` accessors can retrieve the values of `x` and `y`

[6] In our system, the base for every class must be specified explicitly, with `<root>` being the base class in the system.

https://doi.org/10.1017/S0956796814000203 Published online by Cambridge University Press

from a <point-2D> object. The methods point=?, point+, and point- each expect the first argument to be a <point-2D> object and one additional argument, corresponding to the method definition in the <point-2D> class. The biggest syntactic difference between our OOP system and languages like Java or Ruby, is the need to explicitly pass the object to the accessor, mutator, and method procedures.

Classes also support inheritance, so we can define a new class to hold three-dimensional points as follows:

```
(define-class (<point-3D> x y z) (<point-2D> x y)
  (ivars [public z z])
  (methods
    [point=? (o) (and (super o) (= z (<point-3D>-z o)))]
    [point+ (o) (make-<point-3D>
                  (+ x (<point-2D>-x o))
                  (+ y (<point-2D>-y o))
                  (+ z (<point-3D>-z o)))]
    [point- (o) (make-<point-3D>
                  (- x (<point-2D>-x o))
                  (- y (<point-2D>-y o))
                  (- z (<point-3D>-z o)))]))
```

The <point-3D> class inherits from the <point-2D> class. The constructor passes the x and y arguments along to the constructor for <point-2D>. The <point-3D> adds the instance variable z to hold the third dimension of the <point-3D> object in the ivars clause. Since it inherits from the <point-2D> class, the x and y instance variables are also a part of a <point-3D> object, though they are still accessed using the <point-2D>-x and <point-2D>-y accessors, as we can see in the point+ and point- methods. The <point-3D> class also overrides all three of the methods defined in the <point-2D> class, using the super keyword to call point=? from <point-2D> to compare the values in the x and y instance variables before checking the value in the z instance variable.

With the <point-3D> class defined, we can now create <point-3D> objects:

```
(define pt1-3d (make-<point-3D> 1 1 1))
pt1-3d                                  ⇒  #<instance of <point-3D>>
(<point-2D>? pt1-3d)                    ⇒  #t
(<point-3D>? pt1-3d)                    ⇒  #t
(<point-2D>-x pt1-3d)                   ⇒  1
(<point-2D>-y pt1-3d)                   ⇒  1
(<point-3D>-z pt1-3d)                   ⇒  1
(define pt2-3d (make-<point-3D> 13 5 9))
(define pt3-3d (make-<point-3D> 1 1 1))
(point=? pt1-3d pt2-3d)                 ⇒  #f
(point=? pt1-3d pt3-3d)                 ⇒  #t
(point=? pt3-3d pt1-3d)                 ⇒  #t
(define pt4-3d (point+ pt1-3d pt2-3d))
(<point-2D>-x pt4-3d)                   ⇒  14
(<point-2D>-y pt4-3d)                   ⇒  6
(<point-3D>-z pt4-3d)                   ⇒  10
(define pt5-3d (point- pt2-3d pt3-3d))
(<point-2D>-x pt5-3d)                   ⇒  12
```

https://doi.org/10.1017/S0956796814000203 Published online by Cambridge University Press

```
(<point-2D>-y pt5-3d)                    ⇒  4
(<point-3D>-z pt5-3d)                    ⇒  8
```

The <point-3D> objects work similarly to <point 2D> objects with the x and y instance variables accessed through the <point-2D>-x and <point-2D>-y accessors and the z field accessed through the <point-3D>-z accessor. A <point-3D> object can also be treated as a <point-2D> object, as can be seen from how the <point-2D>? predicate responds when given a <point-3D> object. This might not be exactly the functionality we want, however. For instance, it sets up the potentially confusing situation where a <point-2D> object judges a <point-3D> object equal to the <point-2D> object, while the <point-3D> object raises an exception when the point=? method is called with a <point-2D> object.

```
(point=? pt1 pt1-3d)  ⇒  #t
(point=? pt1-3d pt1)  ⇒
Exception in <point-3D>-z: not applicable to #<instance of <point-2D>>
```

The <point-3D> version of the point=? method could be altered to operate the same way as the <point-2D> version of the method, but we may not want a two-dimensional point to ever be equal to a three-dimensional point. However, we might still want to use point=?, point+, and point- as the names of the fields, so that we can build functionality that works over both two-dimensional and three-dimensional points. This is where an interface could be used instead:

```
(define-interface point [point=? (o)] [point- (o)] [point+ (o)])
(define-class (<point-2D> x y) (<root>)
  (ivars [public x x] [public y y])
  (implements point)
  (methods
    [point=? (o) (and (= x (<point-2D>-x o)) (= y (<point-2D>-y o)))]
    [point+ (o)
     (make-<point-2D> (+ x (<point-2D>-x o)) (+ y (<point-2D>-y o)))]
    [point- (o)
     (make-<point-2D> (- x (<point-2D>-x o)) (- y (<point-2D>-y o)))]))
(define-class (<point-3D> x y z) (<root>)
  (ivars [public x x] [public y y] [public z z])
  (implements point)
  (methods
    [point=? (o) (and (= x (<point-3D>-x o))
                      (= y (<point-3D>-y o))
                      (= z (<point-3D>-z o)))]
    [point+ (o) (make-<point-3D> (+ x (<point-3D>-x o))
                                 (+ y (<point-3D>-y o))
                                 (+ z (<point-3D>-z o)))]
    [point- (o) (make-<point-3D> (- x (<point-3D>-x o))
                                 (- y (<point-3D>-y o))
                                 (- z (<point-3D>-z o)))]))
```

Using the point interface, both the <point-2D> and <point-3D> objects can implement the same interface for checking the equivalence of points, adding points, and subtracting points, without having <point-3D> inherit from <point-2D>. It is worth noting that object equivalence is a well known problem in object-oriented programming (Bloch,

https://doi.org/10.1017/S0956796814000203 Published online by Cambridge University Press

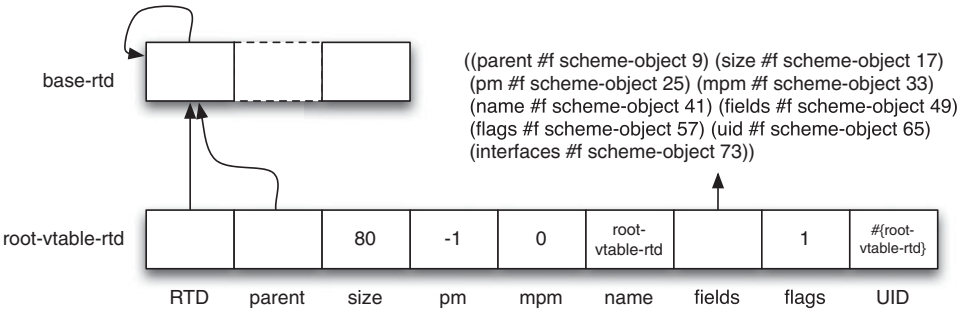

Fig. 12. Root vtable RTD.

2008), and for a given program, we might desire a different notion of equivalence for this case.

9.1.1 Object, class, and interface representations

This contrived example illustrates the use of classes, inheritance, and interfaces in our OOP system. With this understanding in hand, we are ready to discuss the implementation. The OOP system has three types of items: objects, classes, and interfaces.

We would like to keep the representation of our objects as minimal as possible to avoid storing redundant pointers in the object data structure. Since the values of the instance variables are the only things that differ from one object to another object of the same class, it makes sense to store only the values of these instance variables along with some representation of the object's class. This is similar to our record representation where the RTD represents the record type and a record stores a pointer to the RTD along with a slot for each field. However, unlike the record system, we also need a place to store pointers to the implementations of each method that operates on this object, and there is currently no place to store this information in the RTD. In object-oriented languages, this information is often stored in the virtual method table (vtable).

Fortunately, we can extend the base RTD to create a new kind of RTD that can be the basis for our object vtable. We do this by creating a new record that inherits from the base RTD. Figure 12 shows the root vtable RTD. The root vtable RTD adds a new field, `interfaces` to the base RTD fields, since every vtable RTD needs an interfaces field. When a new class is created, a new vtable RTD is created that extends the vtable RTD for the base class. The `<root>` object uses the root vtable RTD as its vtable RTD.

Figure 13 shows the RTD for the first version of the `<point-2D>` class. The vtable RTD for the `<point-2D>` class is a normal record RTD that inherits from the root vtable RTD. It adds the fields `point=?`, `point+`, and `point-` to the fields list of the root vtable RTD. These additional RTD fields are filled with the methods defined for the `<point-2D>` class.

The vtable RTD specifies the fields of an object RTD, and the hierarchy of objects is handled through the record inheritance system in the object RTD's parent field. A class that inherits from the `<root>` class has the special `$instance` record type as the parent RTD. A class that inherits from another class with have that objects RTD as its parent. Thus, in the first example the parent of the `<point-2D>` RTD is the `$instance` RTD, and the parent of the `<point-3D>` RTD is the `<point-2D>` RTD.

https://doi.org/10.1017/S0956796814000203 Published online by Cambridge University Press

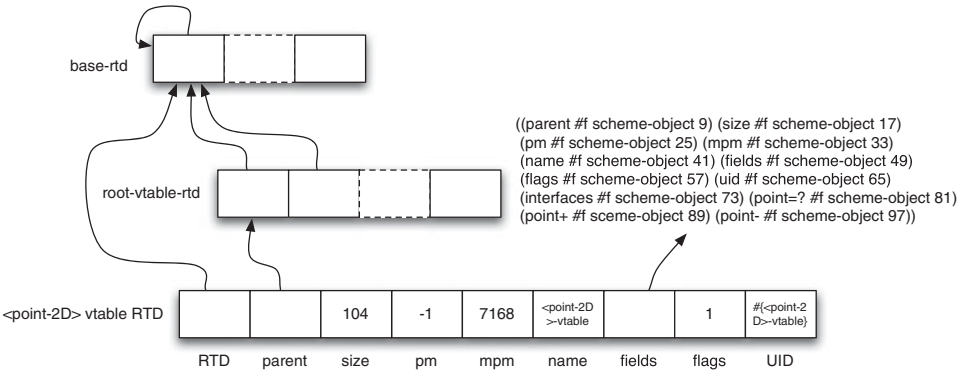

Fig. 13. The `<point-2D>` vtable RTD.

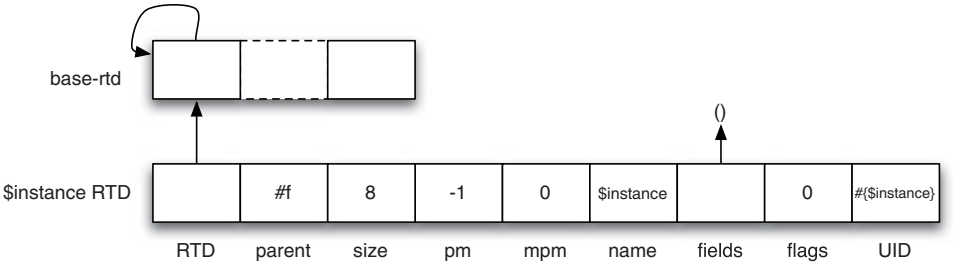

Fig. 14. The `$instance` RTD.

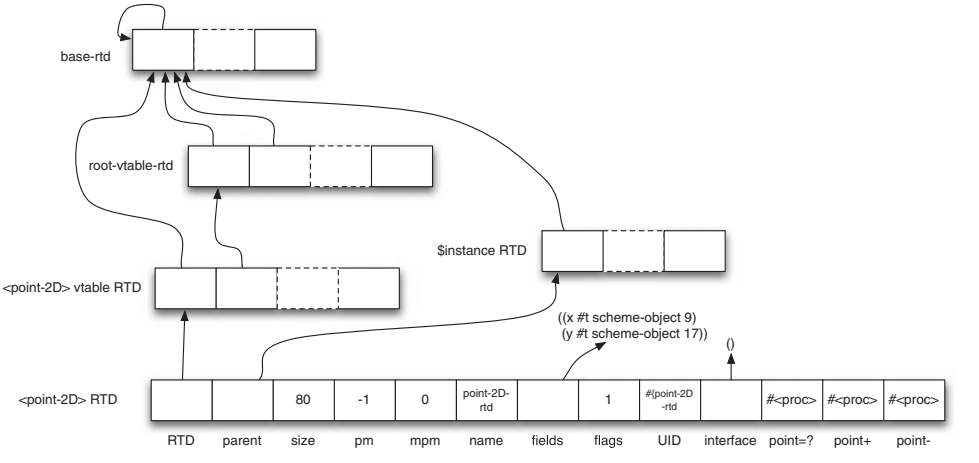

Fig. 15. The `<point-2D>` RTD.

Figure 14 shows the `$instance` RTD. Since the `<root>` class has no instance variables, the field list is empty.

With the `$instance` RTD and the `<point-2D>` vtable RTD, the `<point-2D>` RTD shown in Figure 15 can be created. The `<point-2D>` RTD contains the extra `interfaces` field, along with the `point=?`, `point+`, and `point-` fields. The `interfaces` field stores a list of interface records (discussed later in this section), one for each interface implemented by this class. Since the first `<point-2D>` example does not implement any interfaces,

https://doi.org/10.1017/S0956796814000203 Published online by Cambridge University Press

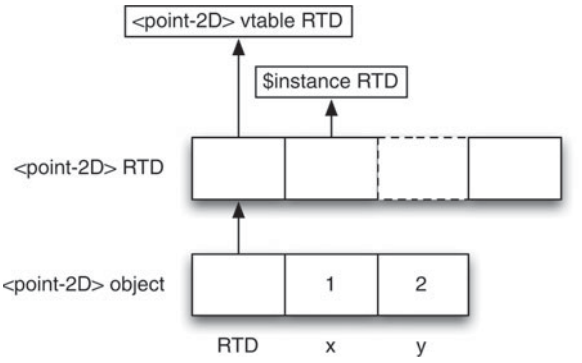

Fig. 16. A `<point-2D>` object.

this field contains an empty list. The `point=?`, `point+`, and `point-` fields in the RTD each hold a pointer to the procedure that implements this method for the class. When one class inherits from another class, the vtable for the new class inherits from the parent class vtable RTD, so the fields for each method are in the child class as well. When the child overrides a method of the parent, it replaces the procedure pointed to by this method entry, with its own procedure. A call to the super method is handled during the expansion of the `define-class` form by looking up the internal name for the super method's procedure.

Figure 16 shows a `<point-2D>` object. Since the methods for the `<point-2D>` object are stored in the vtable RTD for the object, the `<point-2D>` object stores only the values for `x` and `y`. In this case, the object stores the values 1 for `x` and 2 for `y`.

The representation of objects is the most involved part of the OOP system, but a representation for classes and interfaces is still needed. In order to more easily handle inheritance, the `define-class` and `define-interface` forms both store information in the compile-time environment. This information is attached to the identifier used as the name for the class or interface definition. The macros that implement the `define-class` and `define-interface` forms can then look up information for the parent class or interface when they are expanded. This allows the work of determining the full instance variable list, method list, vtable RTD for the base class, and initialization information to be completed at expand time.

A class in the OOP system is represented at compile time by a simple record. The class record type has nine fields, *class formals*, *base formal bindings*, *instance variable bindings*, *method information*, *vtable RTD*, *compile-time RTD*, *vtable expression*, *interfaces*, *initialization procedure*.

The *class formals* field records the formals listed in the class definition following the name. For instance, in the `<point-2D>` class the class formals are `x` and `y`, and in the `<point-3D>` class the class formals are `x`, `y`, and `z`.

The *base formal bindings* field stores the binding of each base class formal to the corresponding base class argument. For instance, in the first `<point-3D>` example, where it inherits from the `<point-2D>` class, the `x` and `y` listed after the base class identifier are the base class arguments, while the `x` and `y` listed in the class definition for `<point-2D>` are the base formals. When a base class itself inherits from a class with arguments, the base class formals include the entire set of bindings.

https://doi.org/10.1017/S0956796814000203 Published online by Cambridge University Press

The *instance variable bindings* contains the full list of bindings between each instance variable of the object and its initial value. This includes the bindings for the instance variables of this class, along with the instance variables inherited from its base class. For instance, the `<point-2D>` class has two instance variable bindings, one binding instance variable `x` to class formal `x` and one binding instance variable `y` to class formal `y`. The `<point-3D>` class that inherits from it has the bindings from `<point-2D>` along with the binding of instance variable `z` to class formal `z`.

The *method information* stores information about methods, including their public name, the arity of the method, the original formals list for the method, along with a flattened list of formals,[7] and an internal name for the method.

The *vtable RTD* is the RTD used to represent the object vtable RTD. It is used as the RTD for the object RTD (as described above).

The *compile-time RTD* is a compile-time representation of the object RTD. It is used to precalculate the offsets of instance variables in the object representation, so that references to instance variables can be rewritten into a low-level reference into the object record. This compile-time RTD can also be used at run time, but only when the class has no methods and no interfaces, which need to be filled out in the RTD at run time.

The *vtable expression* contains the code to build the run-time representation of the object RTD. When there are no methods and no interfaces, this is simply the quoted compile-time RTD. When there are interfaces implemented or member methods of the class, these are recorded in the extra fields of the vtable, as described above.

The *interfaces* field contains the full list of implemented interface identifiers. This includes both the interfaces implemented by this method, as well as those implemented by any ancestor in the inheritance hierarchy.

The *initialization procedure* is the code for a procedure that is called to initialize the fields of a newly created object. This is built by including the initialization code for instance variables with the initialization procedure for the base class.

Like a class, an interface is represented in the compile-time environment using a simple record. The interface record type has two fields: an *interface RTD* field and a *method information* field. The *interface RTD* field stores the RTD for the interface record that is stored in the interfaces field of a vtable RTD for classes that implement this interface. The *method information* field, similar to the class field, stores method name, arity, formals, a flattened formals list, and an internal name for the method.

9.1.2 Class and interface methods

In our OOP system, each method defined by a class or interface has a corresponding Scheme procedure through which the method is invoked. These procedures, however, cannot be redefined each time a class implements an interface or overrides a method, or the previously defined procedure would be lost. Instead, a Scheme procedure is created only when a *new* method is defined by an interface or class. Since a single Scheme procedure might invoke many different object methods, the implementation of the individual methods and the implementation of the procedure are kept separate. The Scheme procedure is,

[7] The formal list may not be a flat list, if this is a variable arity method.

https://doi.org/10.1017/S0956796814000203 Published online by Cambridge University Press

hence, simply a wrapper that determines the method to call from the vtable RTD of the object on which the procedure is invoked. The vtable is represented at compile time using a simple association list, but could use a hash table if the implemented method lists become too long.

The wrapper procedure for a class method tests to make sure that the object it is passed is compatible with this method, then accesses the method in the vtable RTD for the object. Since the vtable RTD for each class is a record that extends the vtable RTD of its parent class, the location of the method to call is at the same memory offset for any shared methods. This is one of the advantages of a single inheritance model. For instance, the implementation for the `point=?` wrapper of the first `<point-2D>` class is implemented as follows:

```
(define point=?
  (lambda (self o)
    (unless (<point-2D>? self)
      (errorf 'point=? "not applicable to ~s" self))
    ((<point-2D>-vtable-point=? (record-rtd self)) self o)))
```

The wrapper procedure for an interface method is slightly more involved, since different classes may implement different sets of interfaces and in different orders. Thus, the first step of the wrapper procedure is to invoke a common procedure used to locate the interface record for this method within the vtable. In the current implementation this is a simple linear search. Once the interface record is located the implementation for the method of this class can be determined and invoked. For instance, the implementation for the `point=?` wrapper of the `point` interface is implemented as follows:

```
(define point-iface-lookup
  (lambda (who self)
    (or (and (record? self)
             (let ([rtd (record-rtd self)])
               (and (root-vtable-record? rtd)
                    (exists
                      (lambda (iface) (point? iface))
                      (root-vtable-record-interfaces rtd)))))
        (error who "not applicable to ~s" self))))
(define point=?
  (lambda (self o)
    ((point-point=? (point-iface-lookup 'point=? self)) self o)))
```

In the interface wrapper method, the `exists` procedure searches for the first matching interface in the interfaces field.

The checks performed in both of the `point=?` procedures can be avoided, if the compiler can determine the check is not needed.

9.2 A representation for foreign structured data

The foreign type or ftype system provides a way to interact with foreign data within Scheme. Ftypes provide a convenient syntactic abstraction for declaring the structure of foreign data, a set of operators for allocating and manipulating that data, and an efficient implementation. The syntax supports all standard C data structures, including struct, union,

https://doi.org/10.1017/S0956796814000203 Published online by Cambridge University Press

array, pointer, function, bit-field, and scalar types, e.g., char, int, double. It also provides a way to define type aliases. Overall, it is reminiscent of C's `typedef`, although it goes beyond `typedef` in allowing the specification of endianness and packing. The fields of an ftype can be accessed individually, or the entire structure can be converted into an S-expression representation for printing or debugging.

Ftypes can be used with the foreign function interface (FFI) to call functions through the application binary interface (ABI). This can be particularly useful when a foreign function takes a pointer to a data structure, or when data is returned through a pointer to a data structure.

We start with a contrived example that demonstrates several ftype forms. The `fun-type` ftype is an example of a function ftype, and the `x-type` ftype includes the full variety of ftype forms.

```
(define-ftype fun-type (function (size_t) void*))

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

The `fun-type` defines a function with a `size_t` argument that returns a `void*`. The `x-type` defines a struct ftype with five fields. The `a` field is a 64-bit union that can be accessed as an array of four 16-bit integers through the `s` field, an array of two big-endian 32-bit integers through the `i` field, or a single little-endian double value through the `d` field. The `b` field is a 16-bit long bit-field split into the 4-bit signed `x` field, 4-bit unsigned `y` field, and 3-bit signed `z` field. It also includes five bits of padding. The `_` syntax is used to represent a field that exists only for padding. Padding is needed because fields of the `bits` form must total 8, 16, 32, or 64 bits. In a C compiler this would be determined automatically, but here we decided to put the control of the bit layout for bit fields in the hands of the programmer, especially since it is unclear what the padding should be if the endianness of the bit-field is non-native. The need to explicitly specify padding up to the size of the container is one way that ftype definitions differ from their C equivalents, where this is managed implicitly. The `c` field specifies a packed struct with a single character field `ch` and an `unsigned-16` field `us`. Since the `c` struct is packed, it is not automatically padded to align `us` on a 16-bit boundary. One byte of padding is explicitly specified to

https://doi.org/10.1017/S0956796814000203 Published online by Cambridge University Press

ensure `us` is aligned. The `f` field is a pointer to a `fun-type`. Finally, the `next` field points to an `x-type`.

Ftype pointers for the `fun-type` and `x-type` are created with `make-ftype-pointer`.

```
(define my-f (make-ftype-pointer fun-type "malloc"))
(define my-x
  (make-ftype-pointer x-type
    (foreign-alloc (ftype-sizeof x-type))))
```

Here, the space pointed to by `my-x` is allocated from Scheme with `foreign-alloc`. The `foreign-alloc` function allocates memory outside the Scheme heap and is similar to C's `malloc` function. An ftype pointer can also be the result of calling a foreign function, or might point to a fixed address in memory, such as one mapped for a device.

The `ftype-pointer?` predicate can identify an object as an ftype and verify that an ftype pointer has a specified ftype.

```
(ftype-pointer? my-f)          ⇒  #t
(ftype-pointer? my-x)          ⇒  #t
(ftype-pointer? x-type my-x)   ⇒  #t
(ftype-pointer? x-type my-f)   ⇒  #f
```

The fields of `my-x` can also be set.

```
(ftype-set! x-type (a d) my-x 2.5)
(ftype-set! x-type (b x) my-x -3)
(ftype-set! x-type (b y) my-x 4)
(ftype-set! x-type (b z) my-x -1)
(ftype-set! x-type (c ch) my-x #\a)
(ftype-set! x-type (c us) my-x 100)
(ftype-set! x-type (f) my-x my-f)
(ftype-set! x-type (next) my-x (make-ftype-pointer x-type 0))
```

Once set, these values can be referenced.

```
(ftype-ref x-type (a i 1) my-x)  ⇒  1088
(ftype-ref x-type (b x) my-x)    ⇒  -3
(ftype-ref x-type (b y) my-x)    ⇒  4
(ftype-ref fun-type () my-f)     ⇒  #<procedure>
```

The `ftype-&ref` operation computes a pointer into an ftype structure, effectively providing controlled pointer arithmetic. The `ftype-pointer-address` procedure can also access the memory address.

```
(define my-int16 (ftype-&ref x-type (a s 3) my-x))
my-int16                              ⇒  #<ftype-pointer #x9DA7B36>
(ftype-ref integer-16 () my-int16)    ⇒  16388
(ftype-pointer-address my-int16)      ⇒  #x9DA7B36
(ftype-pointer-address my-x)          ⇒  #x9DA7B30
```

In addition to accessing elements of `my-x` individually, the entire structure can be converted to an S-expression.

```
(ftype-pointer->sexpr my-x)  ⇒
(struct
  [a (union
       [s (array 4 0 0 0 16388)]
```

https://doi.org/10.1017/S0956796814000203 Published online by Cambridge University Press

```
      [i (array 2 0 1088)]
      [d 2.5])]
  [b (bits
      [x -3]
      [y 4]
      [z -1]
      [_ _])]
  [c (struct
      [ch #\a]
      [_ _]
      [us 100])]
  [f (* (function "__libc_malloc"))]
  [next null])
```

The `ftype-pointer-ftype` operation retrieves the specification of the ftype.

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

The basic run-time unit of the ftype mechanism is the ftype pointer. Two pieces of information are needed for an ftype pointer. One piece is the memory address of the data. The other piece describes the structure pointed to by the address, referred to as the ftype descriptor (FTD). For a scalar ftype, this FTD determines how a value of the type is marshaled, its size in bytes, and its alignment. For structs, unions, arrays, and bit-fields, it contains information about the memory layout of fields and their types. Information about scalar and pointer types allows the implementation to ensure safe access to structures pointed to by an ftype pointer. Information about memory layout allows the implementation to build efficient access and mutation code.

9.2.1 Implementing ftype descriptors

One way to implement an ftype pointer would be to use a Scheme record with two fields: an address field and an FTD field.

```
(define-record-type ftype-pointer (fields address ftd))
```

https://doi.org/10.1017/S0956796814000203 Published online by Cambridge University Press

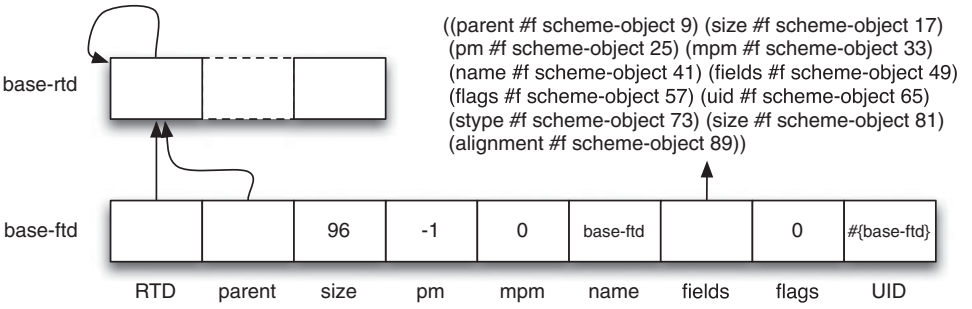

Fig. 17. The base FTD.

In memory, this ftype pointer occupies three fields: an RTD field holding a record type descriptor, an address field, and an FTD field. There are two downsides to this representation. First, ftype pointers need an RTD in addition to the address and FTD, meaning three words of storage are needed. Second, in our system, addresses outside the fixnum range[8] are represented with a bignum. When bignums are used, they must be converted back into a machine-size word before being passed to a foreign function.

Fortunately, just as we extend RTDs to represent OOP-system vtables, we can also extend RTDs to represent FTDs, effectively allowing us to combine two fields into one, and we can use the record system's support for foreign-type fields to represent the address as a raw pointer, eliminating the potential costs of bignum addresses. An ftype descriptor is created by inheriting from `$base-rtd` to create the base FTD. Figure 17 shows the `base-ftd` record. This record extends `$base-rtd` with `stype`, `size`, and `alignment` fields. The `stype` field stores an S-expression representation of the foreign type to support inspection of an ftype pointer. The `size` field is used to indicate the size, in bytes, of the ftype, while the `alignment` indicates if the scalar field is aligned at 8-bit, 16-bit, 32-bit, or 64-bit boundaries.

9.2.2 Representing ftype forms

The `base-ftd` is the root of the ftype system, but FTDs representing struct, union, pointer, array, bits, and scalar types are also needed. Figures 18 and 19 shows the `scalar-ftd` and the `struct-ftd`. The `struct-ftd` extends the `base-ftd` with a `fields` entry to list the items in a foreign struct. Each entry in the `field*` list contains an identifier with the name of the field, the offset of the field from the base pointer, and an FTD indicating the type of the field.

The `scalar-ftd` extends the `base-ftd` with a `swap?` field, used to determine if the byte order of the scalar is swapped for non-native endianness, and a `type` field corresponding to the scalar foreign type it represents, e.g., `integer-32`, `double`, `char`.

The `union-ftd`, similar to the `struct-ftd` extends `base-ftd` with a `field*` entry to represent the items in a union. Fields in the `union-ftd` contain two items: an identifier with the name of the field and an FTD representing the type of the field. Offsets are not needed for union fields, since elements of a union occupy the same space in memory.

[8] Fixnums are 30 bits on 32-bit platforms and 61 bits on 64-bit platforms.

https://doi.org/10.1017/S0956796814000203 Published online by Cambridge University Press

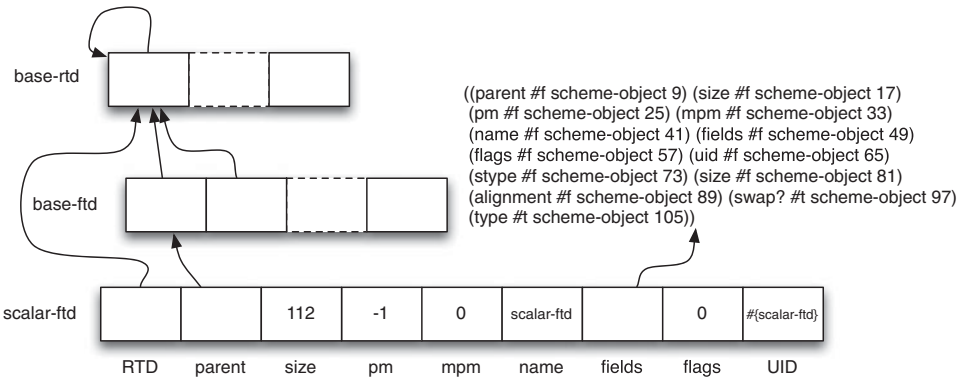

Fig. 18. The scalar FTD.

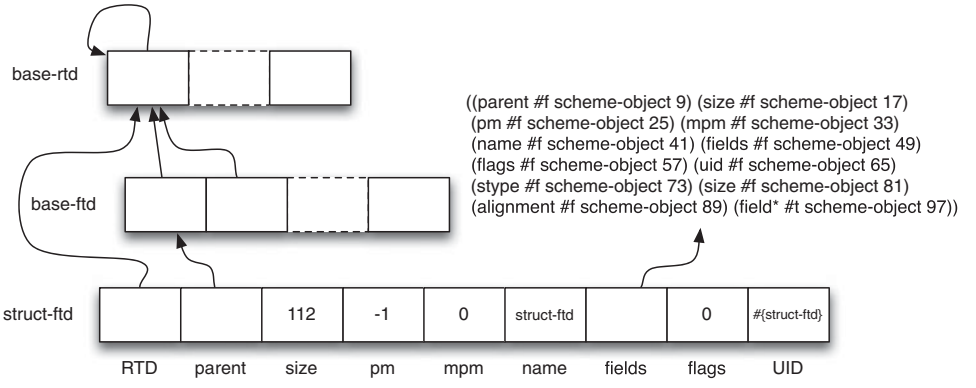

Fig. 19. The struct FTD.

The `bits-ftd` also extends `base-ftd` with a `field*` entry to represent the items in the bit-field. It also adds a `swap?` to indicate if a non-native endianness is used to layout the bytes of the bit-field. Entries in the `bits-ftd` fields list contain four items: an identifier with the name of the field, a flag indicating if the field is signed, the starting bit of the field, and the ending bit of the field.

The `array-ftd` extends `base-ftd` with a `length` field to record the length of the array and an `ftd` field to indicate the type of items in the array.

The `pointer-ftd` extends `base-ftd` with an `ftd` field to indicate the type of the item pointed to. The `ftd` field in a pointer is a mutable field to support ftypes with recursive or mutually recursive structures.

Finally, the `function-ftd` extends the `base-ftd` with three fields: a `conv` field to store the calling conventions, an `arg-type*` field to store a list of argument types, and a `result-type` to store the result type. The `conv` field is only needed to support the Windows operating system, where our system supports the standard calling convention (`__stdcall`), the C declaration convention (`__cdecl`), and the COM calling convention (`__com`). On UNIX-like platforms our system supports only the standard C declaration style calling conventions used by these operating systems, including Linux, Mac OS X,

https://doi.org/10.1017/S0956796814000203 Published online by Cambridge University Press

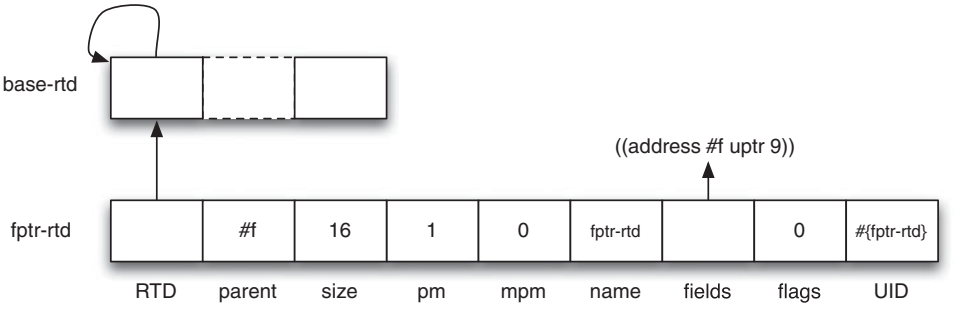

Fig. 20. The base ftype pointer (fptr) RTD.

etc. Together these fields are used by `make-ftype-pointer` to create foreign-callable ftype pointers and `ftype-ref` to create foreign procedures.

9.2.3 Representing `packed`, `unpacked`, and `endian`

The `packed`, `unpacked`, and `endian` forms are not represented by FTDs. Instead, the `packed` and `unpacked` fields determine how the offsets of struct ftypes are calculated. In an unpacked struct (the default), padding is added to ensure fields are placed along machine defined alignment boundaries, as specified by the application binary interface (ABI). In contrast, a packed struct adds no padding, so offsets are calculated based only on field size. This gives programmers full control over the layout of a struct, but care must be taken not to violate the host machine's alignment requirements for retrieving items from memory.

Like the `packed` and `unpacked` forms, the `endian` form is used to determine how an FTD is constructed. If the endianness is different from the native endianness of the machine, a scalar field uses the swapped scalar FTD (one with the `swap?` flag set to `#t`) and a bits field has its `swap?` flag set to `#t`.

9.2.4 Creating ftype pointers

FTDs represent the structure of an ftype pointer, but an ftype pointer is a record with an FTD and a single address field. The `fptr` RTD shown in Figure 20 represents this single-field record. The `fptr` record uses the existing record system's ability to specify the type of its fields to mark the `address` field as a raw, machine-word sized integer. This allows any memory address to be stored in the `address` field without the potential need to use a bignum. Both user defined FTDs and the scalar FTDs inherit from this record.

9.2.5 Scalar ftypes

Each scalar type is represented by a `scalar-ftd` that inherits from the `fptr` RTD. Figure 21 shows the FTD for a scalar double on the Intel x86. It is a scalar FTD, so the RTD field is set to `scalar-ftd` and it inherits from the `fptr` RTD. The `size` and `alignment` fields are both set to 8 on Intel i386 based systems, following the i386 ABI. Size and alignment are set to match the ABI for each supported platform. The double FTD in Figure 21 uses native endianness so the `swap?` field is set to `#f`. The type field is set to `double`. Each

https://doi.org/10.1017/S0956796814000203 Published online by Cambridge University Press

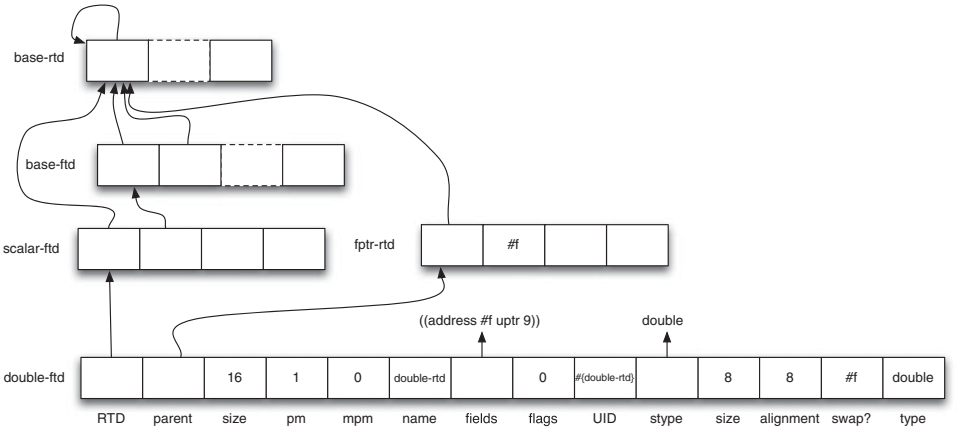

Fig. 21. The double FTD.

scalar type has corresponding swapped and non-swapped FTDs. The swapped versions are used when the endianness differs from the native machine endianness. Scalar FTDs can be broken down into machine-dependent and machine-independent types. The `short`, `unsigned-short`, `int`, `unsigned`, `unsigned-int`, `long`, `unsigned-long`, `long-long`, `unsigned-long-long`, `char`, `wchar`, `float`, `double`, `void*`, `wchar_t`, `size_t`, and `ptrdiff_t` types are machine-dependent and correspond to the like-named C types. The `iptr`, `uptr`, `fixnum`, and `boolean` types are also machine-dependent, where `uptr` is an equivalent to the `void*` type, `iptr` is a signed-integer the same size as the `uptr`, `fixnum` is treated as an `iptr`, but kept in the fixnum range, and `boolean` is treated as an `int` with Scheme `#f` converted to 0, and all other Scheme values convert to 1. The machine-independent types are `single-float`, `double-float`, `integer-64`, `unsigned-64`, `integer-32`, `unsigned-32`, `integer-16`, `unsigned-16`, `integer-8`, and `unsigned-8`, where the `single-float` type corresponds to the 32-bit IEEE single-precision floating point number, the `double-float` corresponds to the double precision 64-bit IEEE floating point number, and the integer and unsigned types represent signed and unsigned integers of the corresponding sizes.

9.2.6 User defined ftypes

When a new ftype is defined a new FTD specifying its structure is also created. For instance, a new struct with two double fields is defined as follows:

```
(define-ftype two-doubles
  (struct
    [a double]
    [b double]))
```

This creates the FTD shown in Figure 22. The RTD is `struct-ftd`, since it is a new type of struct, and it inherits from `double-ftd` since its first element is a `double`. This illustrates how implicit inheritance in ftypes is implemented utilizing the single-inheritance mechanism of the existing record system. The `size` field is 16 bytes, and the alignment field is 8, indicating it must be aligned on an 8-byte boundary. Again, this follows the ABI

https://doi.org/10.1017/S0956796814000203 Published online by Cambridge University Press

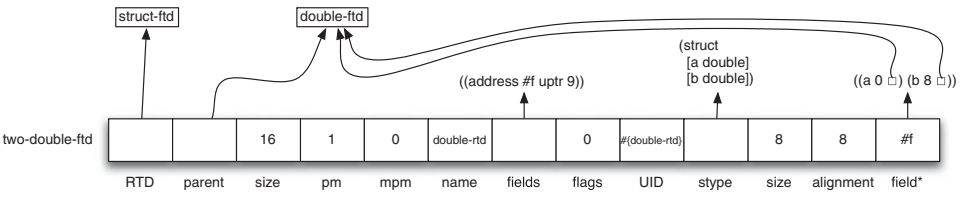

Fig. 22. The FTD for the `two-doubles` ftype.

for the Intel i386. The fields list contain two fields: `a` at offset 0 with FTD `double-ftd` and `b` at offset 8 with FTD `double-ftd`.

9.2.7 Constructing ftype pointers

When an ftype is defined, the FTD is attached to the ftype name identifier. Information about an ftype can be retrieved using this identifier. For instance, `ftype-sizeof` uses this identifier to retrieve the `size`. The `make-ftype-pointer` form retrieves the FTD to use as the RTD of a new ftype pointer. Retrieval operations happen at compile time to avoid run-time overhead, making run-time use of ftype pointers as efficient as possible.

Constructing function ftype pointers requires a little more work since the address expression can be a string or a procedure. When a string is supplied, `make-ftype-pointer` uses the `foreign-entry` function to look up the entry in a shared object (or dynamically-linked library on Windows). The `foreign-entry` function either returns the address of the entry point of the named foreign function or raises an exception indicating it cannot be located. The address is stored in the newly created ftype pointer.

When the address expression is a procedure, a new foreign-callable code object is created. This creates a wrapper function to marshal arguments from foreign values into a Scheme representation before calling the procedure. It also marshals the Scheme return value into a foreign representation. The convention, argument type list, and result type stored in the function FTD along with the host machine ABI specify how this happens. The newly created code object is then locked. This prevents the garbage collector from relocating it when it is expected to be at a given address. Finally, the entry point address of the code object is stored in the pointer position. This allows pointers to foreign-callable Scheme procedures to be passed as part of a larger data structure. When a pointer to the code object is no longer needed, the code object can be extracted via `foreign-callable-code-object` and unlocked with the `unlock-object` function.

9.2.8 Referencing and setting ftype pointers

The `ftype-ref`, `ftype-&ref`, and `ftype-set!` forms allow efficient access to data addressed by an ftype pointer. To ensure this efficiency, these forms look up the FTD for the named ftype at compile time and use the accessors list to determine the offset of data to reference in memory. This means at run time, retrieving the data from memory, can be as fast as two memory references, one to get the base memory address from the ftype pointer, and the second to retrieve the data. More memory references are required if the accessor list includes pointer dereferences, since each deference also needs to be performed to find the data.

https://doi.org/10.1017/S0956796814000203 Published online by Cambridge University Press

For instance, recall the `x-type` example from the start of Section 9.2. In order to access the third `integer-16` stored in the union at the start of the structure we use `ftype-ref` as follows.

```
(ftype-ref x-type (a s 2) my-x)
```

The `ftype-ref` macro retrieves the `x-type` FTD and determines the offset of the third `integer-16` in the `s` array of the `a` union to be 4, since it is 4 bytes from the start of the foreign structure. It also determines the data type of the field is `integer-16`. It produces code to verify that `my-pointer` is an ftype pointer of type `x-type` and a call to the internal `$fptr-ref-integer-16` primitive. This primitive dereferences the address from `my-pointer` and converts the value found there into a Scheme value. The conversion is needed because our fixnum representation differs from the integer representation used by the host machine ABI. In the case of a 16-bit integer, this is a simple shift operation. For larger integers, the size of the integer is checked, and a bignum allocated if the value is too large to store in a fixnum. For instance, when loading a 32-bit integer on a 32-bit platform where the fixnum representation is 30 bits wide, a bignum might be needed. Our system also supports a mode where type checks are disabled. In this mode, the verification step is eliminated.

Referencing a function ftype is a little more involved than referencing scalar or pointer fields. When a function is referenced, a new procedure is created to allow Scheme to call the foreign procedure. A wrapper procedure to marshal Scheme arguments into their foreign representation is created. The wrapper then marshals the foreign return value into a Scheme representation. The convention, argument type list, and result type from the function FTD along with the host ABI are used to determine how this marshaling is handled.

Setting a field in an ftype structure is similar to referencing a field, except that the Scheme value supplied is also checked. If the example above had been an `ftype-set!` operation it would use the `$fptr-set-integer-16` primitive. This primitive checks the value and marshals it into a foreign representation. This type check is eliminated when type checks are disabled. Pointers are also set this way by passing an ftype pointer of the correct type. Only pointer and scalar ftypes can be set.

When the scalar being set or referenced uses an endianness that differs from the native machine endianness, the bytes are swapped appropriately using a machine instruction for byte swapping if one exists or performing the shifts and logical ands necessary if not.

Instead of creating new primitives to handle these operations, the ftype system could have used the existing `foreign-ref` and `foreign-set!` procedures to do this work. The `foreign-ref` procedure references a data in foreign memory offset from a raw pointer address, marshaling a foreign scalar data type into the equivalent Scheme representation. The `foreign-set!` procedure sets foreign memory for a scalar data type offset from a raw pointer address, marshaling a Scheme data type into the equivalent scalar foreign data type. This degrades performance since a raw address is stored in the ftype pointer, and if this value is outside the fixnum range, a new bignum needs to be allocated for it. Further, the `foreign-ref` and `foreign-set!` operations must check to see if the value is a bignum and convert it back to a raw address if it is. Adding new primitives avoided the need for this representation shuffling.

https://doi.org/10.1017/S0956796814000203 Published online by Cambridge University Press

10 Related work

Ikarus Scheme (Ghuloum, 2008), Larceny (Clinger, 2012), and PLT Racket (Flatt *et al.*, 2012), 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. Similar to the way our system uses a common record system to support both R6RS records and our older record system, Racket's implementation of R6RS records also uses the same underlying representation as Racket's structs. In fact, an R6RS record instances created within Racket, responds with `#t` to `struct?`. Racket also includes an object-oriented programming system that extends from the existing struct support. In this way, Racket's struct seems to provide a flexible base similar to our record type system.

Scheme 48 (Kelsey *et al.*, 2008), Chicken (Winkelmann & The Chicken Team, 2012), 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.

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 (Hickey, 2012) supports record-style data types with 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[9] 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 (Ungar & Smith, 1987) 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-oriented programming, and it means that similar to Scheme's

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

https://doi.org/10.1017/S0956796814000203 Published online by Cambridge University Press

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 (Chambers *et al.*, 1989) 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 of a property. The V8 JavaScript implementation (Google, Inc., 2012), 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 objects with the same property name, but different memory layouts, 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) (DeMichiel & Gabriel, 1987) 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 article 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 briefly described in Section 9.2 and in more detail in a previous paper (Keep & Dybvig, 2011).

Smalltalk (Goldberg & Robson, 1983) defines classes using a meta-layer similar to the R6RS record type meta-layer used by our system, and the meta-layer provided by CLOS. Each time a new class is created in Smalltalk, a new metaclass is also created. This is a result of all components in Smalltalk being represented by objects. Thus, the class is itself an object and needs a class, and the metaclass is that class. At the base of this hierarchy is the `Class` class, similar to the `$base-rtd` in our system. In more traditional

https://doi.org/10.1017/S0956796814000203 Published online by Cambridge University Press

Smalltalk-80 systems, metaclasses are created as anonymous classes and treated more as part of the system than user-space objects. However, explicit metaclasses (Briot & Cointe, 1989) have also been experimented with to allow for more flexible changes in the class system.

11 Conclusion

The record type system in Chez Scheme provides an efficient and flexible basis for the R6RS record system. The record type representation determines the memory layout of records when the RTD is created and uses these precalculated offsets when building accessor and mutator procedures. The Scheme object field bit mask and mutable Scheme object field bit masks are also calculated when the RTD is created and allow the garbage collector to determine how fields should be handled, without needing to traverse the list of fields stored in the RTD when a record is swept during the collection process.

Representing RTDs using records provides a record definition meta-layer that allows other record systems, such as our simple OOP system and the ftype system, to be created by extending the base RTD to create new types of RTDs with additional fields. As we saw with the ftype system, the extension that allows foreign data type fields, can be used to represent things like machine-word sized pointers, without the extra overhead needed to create a tagged Scheme-object pointer. This allows the accessors of the ftype system to operate directly on the foreign memory pointer, without the need to convert between a Scheme representation and the foreign representation.

Acknowledgments

This material is based in part on research sponsored by DARPA under agreement number FA8750-12- 2-0106. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

Bloch, J. (2008) *Effective Java (2nd edition) (The Java Series)*. 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall PTR.

Briot, J.-P. & Cointe, P. (1989) Programming with explicit metaclasses in Smalltalk-80. In *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '89. New York, NY, USA: ACM, pp. 419–431.

Chambers, C., Ungar, D. & Lee, E. (1989) 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. New York, NY, USA: ACM, pp. 49–70.

Clinger, W. D. (2012) *Larceny User's Manual*.

DeMichiel, L. G. & Gabriel, R. P. (1987) The common lisp object system: An overview. In: *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '87. London, UK, UK: Springer-Verlag, pp. 151–170.

Dybvig, R. K. (2009) *Chez Scheme Version 8 User's Guide*. Cadence Research Systems.

Dybvig, R. K., Eby, D. & Bruggeman, C. (1994 March) *Don't stop the BIBOP: Flexible and Efficient Storage Management for Dynamically-Typed Lanugages*. Tech. rept. TR400. Indiana University, Bloomington, IN.

Flatt, M., Findler, R. B. & PLT. (2012) *The Racket Guide*.

Ghuloum, A. (2008 October). *Ikarus Scheme User's Guide*.

Goldberg, A., & Robson, D. (1983) *Smalltalk-80: The Language and its Implementation*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Google, Inc. (2012) *Chrome V8: Design Elements*.

Hickey, R. (2012). *Clojure*.

Keep, A. W. & Dybvig, R. K. (2011) Ftypes: Structured foreign types. In *Proceedings of the 2011 Workshop on Scheme and Functional Programming*. Scheme '11.

Keep, A. W. & Dybvig, R. K. (2012) A sufficiently smart compiler for procedural records. In *Proceedings of the 2012 Workshop on Scheme and Functional Programming*. Scheme '12.

Kelsey, R., Rees, J. & Sperber, M. (2008) *The incomplete Scheme 48 reference manual for release 1.8*.

Sperber, M., Dybvig, R. K., Flatt, M., Van Straaten, A., Findler, R., & Matthews, J. (2009) Revised^6 report on the algorithmic language scheme. *J. Funct. Program.* **19**(Supplement S1), 1–301.

Ungar, D. & Smith, R. B. (1987) Self: The power of simplicity. In *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '87. New York, NY, USA: ACM, pp. 227–242.

Waddell, O. & Dybvig, R. K. (1997) Fast and effective procedure inlining. In *Proceedings of the Fourth International Symposium on Static Analysis*, Lecture Notes in Computer Science, vol. 1302. Springer-Verlag, pp. 35–52.

Waddell, O. & Dybvig, R. K. (2004) *A Lightweight Object System for Scheme*. Unpublished manuscript.

Winkelmann, F. L. & The Chicken Team. (2012) *The CHICKEN User's Manual*.