Ftypes: Structured foreign types

Andrew W. Keep   R. Kent Dybvig

Indiana University

{akeep,dyb}@cs.indiana.edu

Abstract

High-level programming languages, like Scheme, typically represent data in ways that differ from the host platform to support consistent behavior across platforms and automatic storage management, i.e., garbage collection. While crucial to the programming model, differences in data representation can complicate interaction with foreign programs and libraries that employ machine-dependent data structures that do not readily support garbage collection. To bridge this gap, many high-level languages feature foreign function interfaces that include some ability to interact with foreign data, though they often do not provide complete control over the structure of foreign data, are not always fully integrated into the language and run-time system, and are often not as efficient as possible. This paper describes a Scheme syntactic abstraction for describing foreign data, a set of operators for interacting with that data, and an implementation that together provide a complete, well integrated, and efficient solution for handling structured data outside the Scheme heap.

1. Introduction

A foreign function interface (FFI) provides a way for programmers using a high-level language, such as Scheme, to call functions written in other languages, through the application binary interface (ABI). This is a convenient way for a language implementor to provide access to existing libraries, in particular system libraries, without providing explicit support for each library. An FFI often allows calls from functions written in other languages into functions written in the high-level language. This allows programs written in a high-level language to interact with functions written in other languages that provide an interface through the ABI, like C, FORTRAN, or higher-level languages. In addition to calling foreign procedures, or allowing procedures to be called from foreign code, a facility for interacting with foreign data is also needed. The high-level language typically represents data differently from the host platform for consistency across platforms and to support garbage collection. In addition to interacting with foreign data created by foreign code, a foreign data facility should also provide access to foreign data resulting from devices on the machine, e.g. DMA data mapped into the address space of a user program. Essentially, a general facility for managing arbitrary data outside the high-level language's heap is desired.

This paper describes a convenient Scheme 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 C data structures, including struct, union, 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 use in Scheme. When accessing scalar elements within an ftype, the value of the scalar is automatically marshaled into the equivalent Scheme representation. When setting scalar elements, the Scheme value is checked to ensure compatibility with the specified foreign type, and then marshaled into the equivalent foreign representation. Ftypes are well integrated into the system, with compiler support for efficient access to foreign data and debugger support for convenient inspection of foreign data.

The ftype syntax is convenient and flexible. While it is similar in some ways to foreign data declarations in other Scheme implementations, and language design is largely a matter of taste, we believe our syntax is cleaner and more intuitive than most. Our system also has a more complete set of features, covering all C data types along with support for packed and unpacked data structures and specifying endianness. The implementation produces efficient code with minimal overhead for accesses and assignments of foreign data, and it appears to be unique in this regard based on systems described in the literature.

The remainder of this paper is organized as follows. Section 2 describes the syntax of ftypes and gives examples of their use. Sections 3 and 4 describe the implementation and its status. Section 5 discusses related work. Section 6 discuses future work and conclusions.

2. Using Ftypes

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. 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 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 memory address can also be accessed with ftype-pointer-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)]
       [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)])
```

2.1 Ftype Syntax

Formally, ftypes are defined using the following syntax.

```
(define-ftype ftype-name ftype)
(define-ftype (ftype-name ftype) ...)
```

The first form defines a single ftype, while the second form allows more than one ftype to be defined simultaneously. The second form allows mutually recursive structures to be defined. An *ftype* is specified by the following grammar.

```
ftype ⟶ manifest-ftype
     | (function conv (arg-type ...) result-type)
     | (function (arg-type ...) result-type)
manifest-ftype ⟶ ftype-name
     | (struct (field-name manifest-ftype) ...)
     | (union (field-name manifest-ftype) ...)
     | (array length manifest-ftype)
     | (* ftype)
     | (bits (field-name signedness bits) ...)
     | (packed manifest-ftype)
     | (unpacked manifest-ftype)
     | (endian endianness manifest-ftype)
```

The `function` form specifies a function type, with a list of argument types in (*arg-type* ...) and the result type in *result-type*. Argument and result types can be pointers to ftypes or scalar ftypes. Passing structs, unions, or functions by value is not currently supported. The optional *conv* specifies the calling conventions. It can be `#f` or a symbol specifying a machine-dependent calling convention. A *conv* of `#f` indicates the default conventions should be used. Function types can appear only at the top level of an ftype definition or within a (`*` *ftype*) form. The `struct`, `union`, and `array` forms correspond to their C equivalents. The `*` indicates a pointer type. The `bits` form specifies a bit field. It differs slightly from a C bit-field, in that the field layout is explicit. The total must be 8, 16, 32, or 64 bits. The `packed` form specifies no padding should be added to fields of a `struct`. This allows programmers to precisely control the layout of an ftype. The `unpacked` form specifies fields of a `struct` should be padded for alignment following the application binary interface (ABI) for the platform. In general, this matches the memory layout used by the C compiler. This is the default when `packed` is not specified. The `endian` form specifies the endianness to use when accessing or setting a field. The *endianness* can be one of `native`, `big`, or `little`.

The `make-ftype-pointer` operation has the following form.

(`make-ftype-pointer` *ftype-name entry-expr*)

The *ftype-name* must be a valid ftype name. The *entry-expr* must be an exact integer representing a memory address, a string, or a procedure. String and procedure arguments are allowed only when the ftype specified by *ftype-name* is a function ftype. When it is a string, `make-ftype-pointer` attempts to look up the entry to a foreign function by this name. If found, the resulting entry is stored as the memory address of the ftype pointer. When *entry-expr* is a procedure, a new foreign-callable code object is created. The code object allows the procedure to be called from foreign code. Once created, the code object is locked, and the address of the foreign entry point is stored in the ftype pointer. This prevents the garbage collector from relocating or removing the code object, allowing foreign code to safely call it at the recorded memory address. Code objects can be retrieved by passing the ftype pointer address to `foreign-callable-code-object` and unlocked with `unlock-object`. This allows the code object to be collected, but it should be re-locked if it is passed to foreign code again.

The `ftype-sizeof` operation returns the size of an ftype. This can be useful when allocating space for a new ftype pointer with `foreign-alloc`.

(`ftype-sizeof` *ftype-name*)

ftype-name must be a valid, non-function, ftype. The size of function ftypes cannot be determined since the function ftype specification does not provide information about the size of the function's code in memory.

Ftype pointers are differentiated from other Scheme objects. The `ftype-pointer?` predicate identifies when an object is an ftype.

(`ftype-pointer?` *expr*)
(`ftype-pointer?` *ftype-name expr*)

The first form is true when *expr* is an ftype pointer. The second form checks that the ftype of *expr* matches the one specified by *ftype-name*.

The `ftype-ref` form references an element of an ftype pointer.

(`ftype-ref` *ftype-name* (*a* ...) *fptr-expr*)
(`ftype-ref` *ftype-name* (*a* ...) *fptr-expr index*)

The *ftype-name* must name a valid ftype. (*a* ...) specifies a list of accessors leading to a scalar, pointer, or function element of the named ftype. The accessors list can include fields named in the ftype, `*` representing pointer dereferencing, or a fixnum-valued expression representing array access. Pointer fields can be treated as arrays using a fixnum expression to reference elements from the start of pointer. Unlike array fields, accessing elements of a pointer cannot be bounds checked, since the number of elements is unknown. Elements of constituent ftypes or elements pointed to by the ftype can also be specified in the accessor list. For instance, the `double` stored in the second element of an `x-type` linked list can be accessed as follows.

```
(ftype-ref x-type (next * a d) my-x)
```

The *fptr-expr* specifies the ftype pointer to be referenced. The *index*, when included, allows an ftype pointer to be treated as a variable length array. Similar to fixnum access of pointer elements, it cannot be bounds checked, since the length of the array is unknown.

When the referenced element is a scalar, `ftype-ref` returns the Scheme representation of the value. For instance, a `char` or `wchar` field returns a Scheme character, a `double` field returns a `flonum`, and an `int` field returns either a `fixnum` or `bignum`, depending on the size of the int. When the referenced element is a pointer, `ftype-ref` returns a new ftype pointer to this element. When a function element is referenced, `ftype-ref` returns a new procedure, allowing the foreign function to be called.

The `ftype-&ref` form is similar to `ftype-ref`, but always returns an ftype pointer. When a scalar or function element is referenced, an ftype-pointer with the element's memory address is created. This can be used to perform pointer arithmetic. When the *index* is specified, `ftype-&ref` treats the *fptr-expr* as a variable length array, returning an ftype pointer to the indexed element.

(`ftype-&ref` *ftype-name* (*a* ...) *fptr-expr*)
(`ftype-&ref` *ftype-name* (*a* ...) *fptr-expr index*)

Values in ftypes can be set using the `ftype-set!` form.

(`ftype-set!` *ftype-name* (*a* ...) *fptr-expr val-expr*)
(`ftype-set!` *ftype-name* (*a* ...) *fptr-expr index val-expr*)

ftype-name, (*a* ...), *fptr-expr*, and *index* are the same as those used by `ftype-ref` and `ftype-&ref`, though only scalar and pointer elements of an ftype can be set. The *val-expr* is the Scheme value to assign to the specified element. The type of the value is checked to ensure it is compatible with the specified foreign type and then marshaled into a foreign representation. If the field specifies non-native endianness, the byte ordering is swapped appropriately.

The `ftype-pointer-ftype` operation retrieves the specification of the ftype, while the `ftype-pointer-address` retrieves the ftype-pointer's memory address.

(`ftype-pointer-ftype` *fptr-expr*)
(`ftype-pointer-address` *fptr-expr*)

The `ftype-pointer->sexpr` operator converts an ftype pointer into an s-expression.

(`ftype-pointer->sexpr` *fptr-expr*)

It uses information about the ftype to determine the structure of the data, marshaling scalar values into Scheme representations, following pointers, and looking up foreign-procedure entry names, when possible. If an invalid pointer is encountered, `invalid` is recorded in the resulting s-expression. If the data is cyclic, a cyclic s-expression will be created that mimics this structure. Generally, this is useful for debugging, but can also be used to create an s-expression copy of the data for Scheme to use, though if this data is to be heavily used, a more efficient representation might be preferred.

2.2 A Networking Example

The C network library uses a variety of structures to support features like converting an IP address to an integer representation or looking up a DNS entry to find its associated IP addresses.

The inet_pton and inet_ntop functions convert a string representation of an IP address into an integer representation and back. Both functions use the in_addr structure, which can be defined as an ftype as follows.[1]

```
(define-ftype in_addr
  (struct
    [addr in_addr_t]))
```

The specification for in_addr_t is as follows.

```
(define-ftype in_addr_t unsigned-32)
```

The in_addr_t is an unsigned 32-bit integer on our host platform.[2] The inet_pton and inet_ntop functions have the following types.

```
(define-ftype pton-t
  (function (int string (* in_addr)) int))
(define-ftype ntop-t
  (function (int (* in_addr) (* char) int) string))
```

Both functions require a pointer to an in_addr structure. Both functions also use the string type. This is a convenience scalar type for functions that work with C's null terminated ASCII strings. In addition to the string type there are also u8*, u16*, and u32* types for representing UNICODE in 8, 16, and 32-bit encodings. These types are not supported directly as ftype scalar types, but can be created by pointers to 8, 16, or 32 bit unsigned integers. An ftype pointer to an in_addr can be created using make-ftype-pointer as follows.

```
(define my-addr
  (make-ftype-pointer in_addr
    (foreign-alloc (ftype-sizeof in_addr))))
```

Scheme procedures to call inet_pton and inet_ntop are created with ftype-ref. The function pointer can be looked up directly using the foreign-entry function, as in the inet-pton definition, or by simply using a string, as in the inet-ntop definition.

```
(define inet-pton
  (ftype-ref pton-t ()
    (make-ftype-pointer pton-t
      (foreign-entry "inet_pton"))))
(define inet-ntop
  (ftype-ref ntop-t ()
    (make-ftype-pointer ntop-t "inet_ntop")))
```

The inet-pton procedure computes the in_addr representation of an IP address.[3]

```
(inet-pton AF_INET "192.168.1.3" my-addr)  ⇒  1
(ftype-ref in_addr (addr) my-addr)  ⇒  50440384
```

The integer representation can be verified with inet-ntop. A pointer to char array is needed to store the result.

```
(define my-char*
  (make-ftype-pointer char
    (foreign-alloc (fx* (ftype-sizeof char) 16))))
(inet-ntop AF_INET my-addr my-char* 16)
 ⇒  "192.168.1.3"
```

The char array my-char* can also be converted into a Scheme string using the following function.

```
(define (char*->string fptr)
  (let f ([i 0])
    (let ([c (ftype-ref char () fptr i)])
      (if (char=? c #\nul)
          (make-string i)
          (let ([str (f (fx+ i 1))])
            (string-set! str i c)
            str)))))
(char*->string my-char*)  ⇒  "192.168.1.3"
```

The char*->string function treats fptr as the pointer to the start of a #\nul-terminated character array, using i to index its elements.

By itself, the in_addr structure is useful for calling inet_pton and inet_ntop. More often it is used as part of the sockaddr_in structure. The sockaddr_in structure is one of the sockaddr family that represents addresses for various types of sockets. These are distinguished by the sa_family_t field at the start of the structure. A sockaddr_common with an sa_family_t field can be defined along with other sockaddr types as follows.

```
(define-ftype sa_family_t unsigned-short)
(define-ftype in_port_t unsigned-16)
(define-ftype sockaddr_common
  (struct
    [family sa_family_t]))
(define-ftype sockaddr
  (struct
    [common sockaddr_common]
    [data (array 14 char)]))
(define-ftype sockaddr_in
  (struct
    [common sockaddr_common]
    [port in_port_t]
    [addr in_addr]
    [zero (array 8 char)]))
```

The sockaddr and sockaddr_in inherit implicitly from the sockaddr_common structure by naming it as the first element of the structure.

The sockaddr is a component of the addrinfo structure. The getaddrinfo function uses this structure to report the IP addresses for a DNS name. It uses the socklen_t field to indicate the size of the sockaddr field.

```
(define-ftype socklen_t unsigned-32)
(define-ftype addrinfo
  (struct
    [flags int]
    [family int]
    [socktype int]
    [protocol int]
    [addrlen socklen_t]
    [addr (* sockaddr_common)]
    [canonname (* char)]
    [next (* addrinfo)]))
```

The addrinfo structure contains information about the address along with a pointer to the next address. The getaddrinfo procedure expects a DNS string, a port or protocol string, a pointer to

[1] Networking structures are machine specific; this example uses the 32-bit Linux version.

[2] Types can be found through man pages, by scanning through header files, through writing a small C program to return size information, or through a tool for defining foreign function interfaces such as FFIGEN or SWIG.

[3] The AF_INET is a machine-specific constant; 2 on Linux.

an addrinfo struct with “hints”, and a pointer to a pointer to to an addrinfo struct. The addrinfo* ftype is defined as a pointer to an addrinfo.

```
(define-ftype addrinfo* (* addrinfo))
(define-ftype getaddrinfo_t
  (function
    (string string (* addrinfo) (* addrinfo*))
    int))
(define getaddrinfo
  (ftype-ref getaddrinfo_t ()
    (make-ftype-pointer getaddrinfo_t
      "getaddrinfo")))
```

Using getaddrinfo the showip example from Beej’s Guide to Network Programming [9] can be adapted into the retrieve-ips Scheme procedure.

```
(define hints
  (make-ftype-pointer addrinfo
    (foreign-alloc
      (ftype-sizeof addrinfo))))

(define addr-char*
  (make-ftype-pointer char
    (foreign-alloc (fx* (ftype-sizeof char) 16))))

(define (retrieve-ips pname)
  (let ([res (make-ftype-pointer addrinfo*
               (foreign-alloc
                 (ftype-sizeof addrinfo*)))])
    (getaddrinfo pname #f hints res)
    (let f ([p (ftype-&ref addrinfo* (*) res)])
      (if (zero? (ftype-pointer-address p))
          '()
          (if (fx= (ftype-ref addrinfo (family) p)
                   AF_INET)
              (cons
                (inet-ntop AF_INET
                  (ftype-&ref sockaddr_in (addr)
                    (cast sockaddr_in
                      (ftype-ref addrinfo
                        (addr) p)))
                  addr-char* 16)
                (f (ftype-ref addrinfo (next) p)))
              (f (ftype-ref addrinfo
                   (next) p)))))))
```

The hints structure is defined outside retrieve-ips, and is reused across calls to the procedure. The family field is set to AF_INET and the socktype field is set to SOCK_STREAM.[4] These settings indicate that IP addresses should be returned by the getaddrinfo function. Assuming the other fields are zeroed out,[5] the two remaining fields can be set as follows.

```
(ftype-set! addrinfo (family) hints AF_INET)
(ftype-set! addrinfo (socktype) hints SOCK_STREAM)
```

The retrieve-ips function allocates space for an addrinfo* to store the result of calling getaddrinfo. Once getaddrinfo is called, retrieve-ips iterates through the linked list pointed to by res. The ftype-&ref operator gets the head of the list from res. The next element is retrieved with ftype-ref at the end of each iteration. In the body of the loop, the network family is checked to verify it is AF_INET and, if it is, the addr field is cast to a sockaddr_in structure. The inet-ntop procedure converts the IP address into a string representation. The ftype-&ref form is used to get an ftype pointer to the in_addr structure to pass to inet-ntop. The cast operator is defined as follows.

```
(define-syntax cast
  (syntax-rules ()
    [(_ ftype fptr)
     (make-ftype-pointer ftype
       (ftype-pointer-address fptr))]))
```

It creates a new ftype pointer using the supplied ftype as the type. This operation is similar to a C cast, except that it allocates a new ftype pointer.

The retrieve-ips procedure can be used to lookup the IP addresses associated with www.google.com.

```
(retrieve-ips "www.google.com")  ⇒
("74.125.225.84" "74.125.225.80" "74.125.225.81"
 "74.125.225.82" "74.125.225.83")
```

2.3 Querying SQLite with a Callback Function

SQLite [16] is a small relational database system with a simple C API for retrieving and setting information in the database. It provides the sqlite3_exec function as a convenient way to execute SQL queries. Data retrieved by this function is returned via a callback procedure.

Before running the example, a test.db file is setup using the sqlite3 command line program with the following commands.

```
sqlite3 test.db
sqlite> CREATE TABLE example
          (id INTEGER PRIMARY KEY,
           name VARCHAR(50),
           email VARCHAR(50));
sqlite> INSERT INTO example (id, name, email)
        VALUES (1, "Andrew W. Keep",
          "akeep@cs.indiana.edu");
sqlite> INSERT INTO example (id, name, email)
        VALUES (2, "R. Kent Dybvig",
          "dyb@cs.indiana.edu");
```

This example demonstrates using ftypes to create a callback procedure in Scheme to use with sqlite3_exec. The sqlite3_open and sqlite3_close functions are also needed to open and close the database.

These functions use an opaque data structure as a database handle. The sqlite3_open function uses a pointer to a database handle to setup the database. The sqlite3_exec function also needs a callback function type and a pointer to a char pointer type to return error messages. Ftypes for these are defined below.

```
(define-ftype db-handle void*)
(define-ftype db-handle* (* db-handle))
(define-ftype cb-t
  (function (void* int (* char*) (* char*)) int))
(define-ftype char* (* char))
```

Types for sqlite3_open, sqlite3_close, and sqlite3_exec are also needed.

```
(define-ftype open-t
  (function (string (* db-handle*)) int))
(define-ftype close-t
  (function ((* db-handle)) int))
```

[4] SOCK_STREAM is a machine-dependent constant; 1 on Linux.

[5] Fields could be zeroed out individually, or all at once with C’s memset function.

```
(define-ftype exec-t
  (function
    ((* db-handle) string (* cb-t)
     void* (* char*))
    int))
```

These function types can be used with `ftype-ref` to create the `db-open`, `db-close`, and `db-exec` procedures.

```
(define db-open
  (ftype-ref open-t ()
    (make-ftype-pointer open-t "sqlite3_open")))
(define db-close
  (ftype-ref close-t ()
    (make-ftype-pointer close-t "sqlite3_close")))
(define db-exec
  (ftype-ref exec-t ()
    (make-ftype-pointer exec-t "sqlite3_exec")))
```

The `cb-t` callback expects four arguments: a `void*`, an int, and two pointers to `char*`. The first argument allows an arbitrary pointer to be passed from `sqlite3_exec` to the callback. This field is unused in this simple example. The second argument indicates how many values and columns of data are returned. The third and fourth arguments are variable length arrays of `char*` arguments containing the column names and values. The `callback` procedure loops `cnt` times over the `vals` and `cols` arrays, printing out the column name and value. It makes use of `char*->string` from the networking example to convert a `char*` into a Scheme string. When finished, it returns 0 to indicate success. The `cb` ftype pointer is setup with the `callback` procedure to to prepare for the `db-exec` call.

```
(define (callback ignore cnt vals cols)
  (do ([i 0 (fx+ i 1)])
      ((fx= i cnt))
    (printf "~a: ~a\n"
      (char*->string (ftype-ref char* () cols i))
      (char*->string (ftype-ref char* () vals i))))
  (printf "\n")
  0)
(define cb (make-ftype-pointer cb-t callback))
```

A call to the `db-open` function opens the database file and initializes a handle to the database for use in the call to `db-exec`. `db-open`.

```
(define db*
  (make-ftype-pointer db-handle*
    (foreign-alloc (ftype-sizeof db-handle*))))
(db-open "test.db" db*)
```

The `db-exec` function also needs a pointer to a `char*` to return error messages. Once tese are setup, the `db-exec` function can be called with the database handle, an SQL expression, the callback, a 0 for the ignored data pointer, and `zerr` for catching error messages.

```
(define zerr
  (make-ftype-pointer char*
    (foreign-alloc (ftype-sizeof char*))))
(db-exec (ftype-ref db-handle* () db*)
  "SELECT * FROM example;" cb 0 zerr)  ⇒
id: 1
name: Andrew W. Keep
email: akeep@cs.indiana.edu

id: 2
name: R. Kent Dybvig
email: dyb@cs.indiana.edu
```

The `callback` procedure is called once for each result row, and passed a value and column name for each of the three columns. The data inserted at the `sqlite3` command line is returned through `callback`.

Finally, the code object from the `cb` variable is unlocked and the database closed.

```
(unlock-object
  (foreign-callable-code-object
    (ftype-pointer-address cb)))
(db-close (ftype-ref db-handle* () db*))
```

Here, `foreign-callable-code-object` is passed the memory address for `cb`, retrieving the associated code object, and in turn passing it to `unlock-object`. The database is closed using `db-close`, called with the database handle.

2.4 Debugging with Ftypes

Making ftypes a well integrated part of the system is another important goal. This means providing pretty printing and debugging support to treat an ftype like any other Scheme type.

The `ftype-pointer->sexpr` operation uses information stored with the ftype pointer to determine the names and types of ftype fields. It can convert complicated foreign structured data into an s-expression.

For instance, the `ftype-pointer->sexpr` made it easy to see the structure returned by `getaddrinfo` while developing the code for the network example.

```
(getaddrinfo "www.google.com" #f hints res)
(ftype-pointer->sexpr res)  ⇒
(* (struct
     [flags 0]
     [family 2]
     [socktype 1]
     [protocol 6]
     [addrlen 16]
     [addr (* (struct [family 2]))]
     [canonname null]
     [next
      (* (struct
           [flags 0]
           [family 2]
           [socktype 1]
           [protocol 6]
           [addrlen 16]
           [addr (* (struct [family 2]))]
           [canonname null]
           [next ---
            (* (struct
                 [flags 0]
                 [family 2]
                 [socktype 1]
                 [protocol 6]
                 [addrlen 16]
                 [addr (* (struct [family 2]))]
                 [canonname null]
                 [next null]))])]))])))
```

The contents of the `res` ftype pointer is turned into an s-expression, down to the final terminating `null`. This example shows only three levels of the returned `addrinfo` structure (the first two and the last one) to allow it to fit on the page, indicating the elided region with `---`. The final item in the list is terminated by a NULL pointer, indicated by `null` in the `next` field.

When dealing with data containing raw pointers, it is possible that a pointer is valid, but the data does not conform to the shape expected by the ftype description, either because it is not yet initialized or because data of a different type has been written to this position in memory, have a pointer to an `in_addr` structure where the `next` pointer has been overwritten with an invalid pointer by some other operation. In this situation, an unexpected s-expression might be generated. To support debugging these types of issues, something more powerful than `ftype-pointer->sexpr` is needed. Hence, the inspector also supports inspecting ftypes. The inspector is available both from the Scheme REPL and the Scheme debugger.

It can traverse ftype data structures, print values stored in memory, or alter foreign data. For instance, the `getaddrinfo` is replaced with a user specified function and the `retrieve-ips` loops indefinitely, the `addrinfo` data structure can be inspected. If, while traversing the data structure, the list is found to be unintentionally circular, the `next` pointer at the end of the list can be set to NULL through the inspector, and debugging can continue to ensure that there are not other problems with `retreive-ips`.

3. Implementation

An efficient implementation is one of the primary goals of the ftype mechanism, and it informed the design of the ftype syntax. The ftype system leverages the fact that ftype pointers are accessed through syntactic forms to calculate as much as possible about the memory offsets at compile time, leaving a minimum of work to be done at run time. This along with an efficient representation of ftypes in memory and a set of open-coded compiler primitives helps generate as few instructions as possible.

3.1 Ftype Pointers

The basic run-time unit of the ftype mechanism is the ftype pointer. Two pieces of information are needed for an ftype pointer. The first piece describes the structure of the ftype pointer. For scalar ftypes, this information determines how values are marshaled, its size in bytes, and its alignment. For structured types, 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. This is referred to as the ftype descriptor (FTD). The second piece of information is the memory address of the data.

3.2 Representing Ftype Descriptors

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

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

In memory, ftype pointers would occupy three fields: an RTD field holding a record type descriptor, an ftd field, and an address field. There are two downsides to this representation. First, ftype pointers need an RTD in addition to the FTD and address, meaning three words of storage are needed. Second, in our system, addresses outside the fixnum range[6] 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.

3.3 Extending the Record System

Fortunately, the existing record system supports both of these goals. A record with a single field, such as the one defined below, is similar to our vision of the ftype pointer.

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

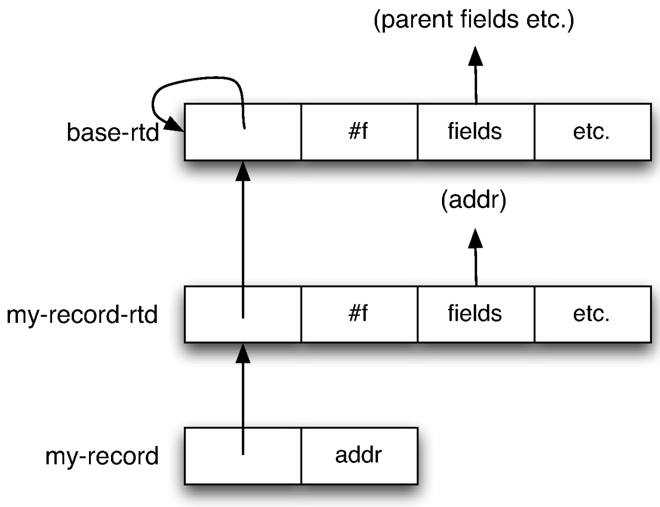

Figure 1. The base RTD, the RTD for `my-record`, and an instance of `my-record`

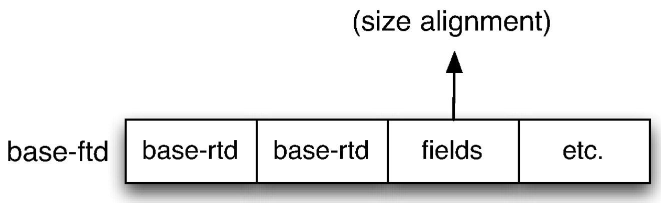

Figure 2. The base FTD

```
(define-record-type my-record (fields addr))
```

A record with a single field occupies two words of memory. The first word is an RTD and the second word stores the value of the field. This is shown in Figure 1 as the object labeled `my-record`.

RTDs are represented as records. An RTD record has a parent field, to support inheritance, a field listing the fields of the record, and other information beyond the scope of this discussion. The RTD for `my-record` is shown in Figure 1 as `my-record-rtd`.

The RTD for this record is the `base-rtd`, since it is a record type descriptor. The `#f` in the parent field indicates it does not inherit from other record types. The `fields` element of the `my-record` RTD lists the single `addr` field.

The `base-rtd` is also represented in Figure 1. Since it is a record type descriptor, it has itself as its RTD. The `#f` in the parent field indicates it has no parent. In its `fields` entry it lists the fields necessary to create new RTDs.

Representing RTDs with records, allows the record system to be extended by creating new kinds of RTDs that inherit from the `base-rtd`. The ftype descriptor is created by inheriting from the `base-rtd` to create the base FTD. Figure 2 shows the `base-ftd` record. This record extends `base-rtd` with `size` and `alignment` fields. The `size` field is used to indicate the size, in bytes, of the ftype, while the `alignment` indicates how it should be aligned in memory. The `base-ftd` allows ftype pointers to be stored in two words, satisfying the first part of the desired ftype pointer structure.

3.4 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

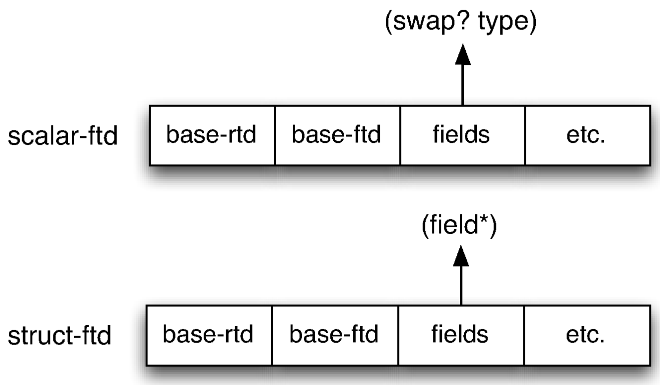

Figure 3. The scalar and struct FTDs

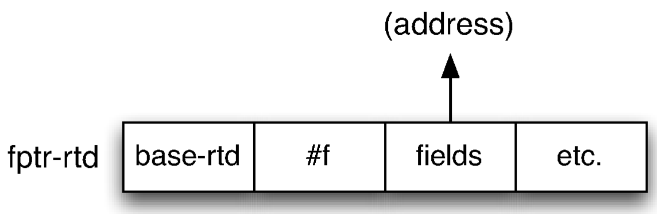

Figure 4. The base ftype pointer (fptr) RTD

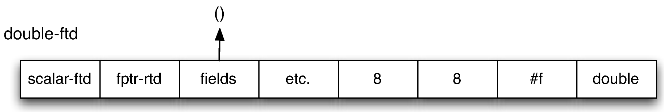

Figure 5. The double FTD

needed. Figure 3 shows the `struct-ftd` and the `scalar-ftd`. The `struct-ftd` extends the `base-ftd` with a `fields` entry to list the items in a foreign struct. Each entry in the `fields` 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 `fields` 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.

The `bits-ftd` also extends `base-ftd` with a `fields` 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. These are used by `make-ftype-pointer` to create foreign-callable ftype pointers and `ftype-ref` to create foreign procedures.

3.5 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 id different from the native endianness of the machine then a scalar field will use the swapped scalar FTD (one with the `swap?` flag set to `#t`) and a bits field will have its `swap?` flag set to `#t`.

3.6 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 4 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 need to use a bignum, satisfying our goal of not creating bignums for addresses. Both user defined FTDs and the scalar FTDs inherit from this record.

3.7 Scalar Ftypes

Each scalar type is represented by a `scalar-ftd` that inherits from the `fptr` RTD. Figure 5 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 5 uses native endianness so the `swap?` field is set to `#f`. The type field is set to `double`. Each 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

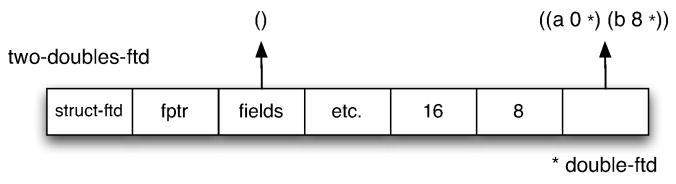

Figure 6. The FTD for the `two-doubles` ftype

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.

3.8 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 6. 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 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`.

3.9 Constructing ftype pointers

When ftypes are 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` information. The `make-ftype-pointer` form retrieves the FTD to use it as the RTD of a new ftype pointer. These operations happen at compile time to avoid run-time overhead, making 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 returns either the address of the entry point of the named foreign function or raises an error 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 found using `foreign-callable-code-object`, and then unlocked with the `unlock-object` function.

3.10 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.

For instance, recall the `x-type` example from Section 2. In order to access the third `integer-16` stored in the union at the start of the structure we would 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 happens.

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. 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 would need 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.

4. Status

The ftype system described in this paper is implemented in Chez Scheme Version 8.4 [5], which runs on a variety of platforms. Unpacked ftypes on each platform follow the ABI for the platform. This ensures that unpacked structures match those produced by the C compiler and that the alignment of structs, unions, and scalar types also match those produced by the C compiler. Since the ABI on host platforms do not frequently change, the implementation is isolated from problems of interacting with new compilers for C, FORTRAN, or other high-level languages that support the host ABI.

A large test suite exists to test that the ftypes system follows these conventions. Beyond testing basic functionality, ftypes produced by the Scheme system are compared with equivalent definitions created by the C compiler. The tests create both ftypes and C structures and then compile C code that reports the size and offsets of fields to compare with the equivalent ftypes. This has helped verify, within the limits of testing, that our understanding of the ABI is correct, and helped identify places where we have interpreted them differently from the C compiler. The test suite helped us discover some inconsistencies between our implementation and the platform's C compiler. These have been resolved on all platforms, except for one obscure inconsistency on Mac OS X PowerPC, where the C compiler does not appear to adhere to the ABI. Since this platform is no longer supported by Apple, we have decided to leave this relatively obscure bug.

5. Related Work

Many aspects of the ftype system exist in other Scheme foreign function interface systems. The most common feature to support is C-style structs, but most other structural features are supported by at least one other system, with the exception of bit-fields. Support for endianness also appears to be unique to the ftype system. On the implementation side, it is more difficult to know how closely other systems match our own as the low-level details of these features tend not be documented, but we believe our overloading of record type descriptors to double as ftype descriptors defining the structure of foreign data is unique, as well as the efficiency of the resulting code. One feature common in other systems that is not included in the ftype system is support for C-style enumerations. Since these are essentially integers, and the C standard does not require any additional checking be done for them, we have decided to leave them out of the system, though a macro to define enumeration types is straightforwardly defined:

```
(define-syntax (define-enum-ftype x)
  (define (enum e*)
    (let f ([e* e*] [n 0])
      (if (null? e*)
          '()
          (syntax-case (car e*) ()
            [(e v)
             (cons #'(e v)
               (f (cdr e*)
                  (+ (syntax->datum #'v) 1)))]
            [e (identifier? #'e)
             (cons #`(e #,n)
               (f (cdr e*) (+ n 1)))]))))
  (syntax-case x ()
    [(_ name e* ...)
     (with-syntax ([((e v) ...) (enum #'(e* ...))])
       #'(begin
           (define-ftype name int)
           (define e v) ...))]))
```

Building support for enumerations into the ftype system would allow for additional checking that the C standard allows but does not require.

5.1 Scheme Foreign Data

Racket uses libffi [1, 8] to support the foreign function interface. Beyond the basic support provided by libffi, Racket provides a type system for foreign types. Similar to the ftype system, these types can be used when looking up FFI objects such as foreign procedures. Beyond support for the basic scalar types and a variety of string types, the type system can be extended with both C style structs and tagged pointers. Like the ftype system, structs defined with the `define-cstruct` syntax create a new pointer type that can be used as either an argument or return value to a foreign procedure. Structs provide a set of mutator functions that verify the Scheme value provided is of the correct type and marshal it into the appropriate C type, similar to the `ftype-set!` form. Structs created this way also have a tagged pointer type and support both implicit inheritance (when the first field of a struct is a struct) and explicit inheritance by naming a struct to inherit from. Tagged pointers also support type inheritance and pointer arithmetic. Racket also includes support for function types, enumerations, and an enumeration type based on bitmasks. Racket provides a simpler syntax for specifying function argument types than the ftype system, partially because the ftype system currently needs to attach the function type to a name. Finding a more flexible way to create function ftype pointers in place of the current `make-ftype-pointer` syntax would help to alleviate some of the difficulties with the ftype system.

Larceny Scheme provides a layered FFI [12], with memory peek and poke operations supporting low-level access to foreign memory. These operations support a set of extensible core scalar, pointer, function, maybe, and one-of types. Scalar types are extended using `ffi-add-attribute-core-entry!` with an existing type, along with a function to marshal values from Scheme into C and optionally a function to marshal values from C back into Scheme. New pointer types, based on the `void*` type can also be created and support a hierarchical type structure. Pointers can be cast up or down the hierarchy, but are checked to ensure they are never cast across the hierarchy. These new types can be used as parameters or return types to functions.

At a higher level, Larceny also provides the `define-c-info` macro to look up information about C constants, structures, etc. This macro looks up information by creating and running a small C program during expand time to determine information such as constants or struct sizes and offsets. This retrieves machine-specific information without requiring the programmer to look it up on each supported platform. We see this feature as complementary to the ftype system, which could use information from a generated C program at expand time to create an appropriate ftype definition. This feature does require a C compiler be available during expansion time, but once expanded, code can be compiled on machines without a C compiler.

At a higher level, a `define-c-struct` macro utilizes `c-info` to create a constructor that builds an appropriately sized bytevector for the struct. This bytevector is allocated in non-relocatable memory, so it can be safely passed to a C function as a boxed argument, but cannot be used as a return value for a function. In this way it is not as flexible as the ftype system.

On the implementation side, pointers that derive from `void*` are stored similar to our ftype pointers, in that a record RTD is used to determine the type and inheritance of the pointer. No structural information about the data is provided. Foreign data accesses are currently done through a low-level peek and poke, which reads or writes to a bytevector before marshaling data to the appropriate type [13]. As such, we believe our accesses to be a little faster, though primitives similar to our own could certainly be added to Larceny.

Bigloo Scheme [15] also provides support for exporting struct, union, array, pointer, enum, function, and opaque types, when compiling to C. Structs and unions provide a constructor, a predicate, a set of accessors, and a set of mutators that make use of Bigloo's optional type specifiers. These functions serve in place of `make-ftype-pointer`, `ftype-pointer?`, `ftype-ref`, and `ftype-set!`. These definitions include the underlying C names, which Bigloo can make effective use of through its compilation into C code. Opaque types are defined using this information from C to handle types when the internals are not needed by the Scheme code. The syntax is similar to the ftype syntax, though nested types must be named since the type and field name are tied together in the syntax. Bigloo also provides a tool called `Cigloo` that can be used to produce C data types from header files.

Gambit Scheme [6] also provides an interface for defining structs, unions, and general types. These forms are specified without field information, relying on the information gained from the C compiler, once Scheme code is compiled into C. Since the structure is not provided explicitly, the programmer must also provide marshaling functions for foreign data. Gambit also provides a mechanism for defining function pointer types.

5.2 Foreign Data in Common Lisp

Common Lisp provides two main standards for implementing a foreign function interface, the universal foreign function interface (UFFI) [14] and the common foreign function interface (CFFI) [3]. Both standards define a way to define structs, unions, and pointers. In both cases these define new types that can be used in the definition of foreign functions similar to the ftype system. Both standards also allow for arbitrarily nested structures. UFFI uses a similar syntax to ftypes, allowing definitions of arrays as fields of the structs and unions. Elements of structs are accessed through the `foreign-slot-value` operator in CFFI or the `get-slot-value` operator in UFFI, which have a similar form to the `ftype-ref` form.

5.3 ML Foreign Data

Standard ML of New Jersey (SML/NJ) provides a method for encoding C types as ML types [11] and working with data defined in these structures. Foreign data created with this facility is stored in the ML heap along with other ML data. When ML makes a foreign call this data is marshaled into a C representation and copied into the C heap. When C calls back into SML/NJ procedures, this data is then marshaled back into ML representations and copied into the ML heap. Efficient access to foreign data is not the primary goal of this solution, and in this way, it differs from the ftype system.

A second foreign function interface (FFI) for SML/NJ [4] has been proposed and implemented. Like the current SML/NJ FFI, this version also encodes C types using the ML type system, but instead of copying data between the ML and C heaps, it extends the existing SML/NJ system to add primitive operations to deal with C data representations. Like the ftype system, it attempts to encode the full C data representation semantics. Also like the ftype system, the implementation of this encoding of C data attempts to be as efficient as possible. It also includes structural equality of C-types, something the ftype system does not support.

5.4 Interface Generation Tools

While these programming implementations and others allow programmers to define interfaces to foreign libraries and programs, it can be a tedious process to implement such an interface. This process is ripe for automation since it tends to be driven from declarations of constants, data structures, and functions in C header files. Several tools exist to help ease the process of generating these interfaces.

In the Scheme community, FFIGEN [10] can be used to generate foreign function interfaces for a variety of Scheme implementations. FFIGEN is divided into a front-end that utilizes LCC [7] to parse C header files and generate an internal representation for constants, data structures, and function definitions. This internal representation can be used with one of the Scheme implementation specific back-ends to generate the Scheme code to interact with these libraries.

In the broader community, the SWIG project [2] provides a similar interface, but supports a much wider range of languages, and can also be used to generate C stubs, if required by the host language. Similar to FFIGEN it is divided into a front-end that effectively acts as a C parser and compiler and a set of back-ends used to generate code for the various languages supported. SWIG already provides support for several Scheme implementations, in addition to the other programming languages.

Tools that generate interface code, such as these, are complementary to the purpose of the ftype system.

6. Conclusions and Future Work

The ftype system provides a convenient way to manage data outside the Scheme heap, including C-style data structs, unions, arrays, and bitfields as well as memory whose arbitrary structure is determined by the Scheme programmer or the requirements of interacting with low-level devices and device drivers. The implementation is efficient, with accesses boiling down to inline machine code that performs a type-check (when type-checking is enabled), two memory references, and a simple marshaling operation. It is well integrated into our compiler and run-time system, including the debugger, which understands ftypes and allows the printing, inspection, and modification of arbitrary foreign data. With type checking enabled (the default), use of ftype pointers is type safe to the extent that the ftype declaration is a faithful description of the underlying data. In other words, as long as the address provided is a valid instance of the ftype, all accesses from then on are safe. We believe ftypes provide a good model for interfacing with foreign data, in a reasonably safe, efficient, and convenient manner.

Currently, each occurrence of an ftype definition in the source code declares a new ftype distinct from all other ftypes, including those with similar structure. This avoids potentially costly structural comparisons at run time, but a method for allowing the same ftype to be defined in multiple places would be useful. One approach would be to add a `nongenerative` ftype form that is similar in behavior to the nongenerative `define-record-type` subform, although a mechanism that treats all syntactically similar ftype definitions as identical might be nice if the amortized cost of the run-time type equivalence checks can be made low enough.

In the current ftype system, a struct is an implicit subtype of the type of its first field, and an array ftype is an implicit subtype of the type of each element. From a purely functionality perspective, a union ftype should similarly be an implicit subtype of each of its members. We have chosen not to do this because the underlying implementation supports only single inheritance and the added type-checking overhead might not be justified in any case. It might be nice to find an efficient way to perform type checks for union types or to provide a syntax for explicit inheritance from some or all of the member types.

We have not yet implemented a mechanism for automatically generating ftypes from C header files using a tool like FFIGEN [10] or SWIG [2]. Since both of these tools allow for the development of new back-ends, it should be straightforward to add a back-end that supports ftypes. We might also want to provide a tool like Larceny's `c-info` system that can be used to determine information via a C compiler, if available, at expand time.

Acknowledgments

Thanks to members of the Corporate Development Technology group at Cisco Systems for early feedback and discussions about these extensions. Comments from the anonymous reviewers led to several improvements in this paper's presentation.

References

[1] E. Barzilay. Foreign interface for plt scheme. In *Workshop on Scheme and Functional Programming*, 2004.

[2] D. M. Beazley. SWIG : An easy to use tool for integrating scripting languages with C and C++. *Proceedings of the 4th USENIX Tcl/Tk Workshop*, July 1996.

[3] J. Bielman and L. Oliveira. CFFI user manual, August 2010. URL `http://common-lisp.net/project/cffi/manual/index.html`.

[4] M. Blume. No-longer-foreign: Teaching an ML compiler to speak C. *Electronic Notes in Theoretical Computer Science*, 2001.

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

[6] M. Feeley. Gambit-C user manual, March 2011. URL `http://www.iro.umontreal.ca/~gambit/doc/gambit-c.html`.

[7] C. W. Fraser and D. R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0805316701.

[8] A. Green. The libffi home page. URL `http://sourceware.org/libffi/`.

[9] B. Hall. *Beej's Guide to Network Programming*. Jorgensen Publishing, February 2009.

[10] L. T. Hansen. FFIGEN user's manual, February 1996. URL `http://www.ccs.neu.edu/home/lth/ffigen/userman.html`.

[11] L. Huelsbergen. A portable C interface for Standard ML of New Jersey. Technical report, AT&T Bell Laboratories, January 1996.

[12] F. Klock II. The layers of Larceny's foreign function interface. *Scheme and Functional Programming Workshop*, Sept. 2007.

[13] F. Klock II. Email correspondence. private communcation, 2011.

[14] K. M. Rosenberg. UFFI reference guide, 2003. URL `http://uffi.b9.com/manual/`.

[15] M. Serrano. Bigloo: A practical Scheme compiler user manual for version 3.6a, January 2011. URL `http://www-sop.inria.fr/mimosa/fp/Bigloo/doc/bigloo.html`.

[16] The SQLite authors. The SQLite home page, 2011. URL `http://www.sqlite.org/`.