# Integrating the Scheme and C Languages

**John R. Rose, Hans Muller**
*SunPro, a Sun Microsystems Business*
*JRose@Eng.Sun.COM, HMuller@Eng.Sun.COM*

**April 10, 1992**

Most implementations of Scheme (and other Lisp dialects) provide some facility for calling functions defined in ANSI C (or other popular languages) even if only to implement Scheme's primitive operations. Some relatively sophisticated implementations provide access to non-Scheme data structures. Taking a Scheme-centered view, we will refer to these facilities as *foreign call-out*, for both data and functions access. Scheme implementations may also provide ways for C code to use Scheme functions and data structures more or less directly. We will refer to this as *foreign call-in.*

Call-in is usually more difficult than foreign call-out, because Scheme systems depend on a strong set of invariants relating to storage management, safety, and runtime dispatching, and these invariants must be restored and maintained by foreign code which makes call-ins.For these reasons, many Scheme systems are not packaged as libraries. Rather, they are full environments which must take over the management of the entire address space in which they run.

In practice, many foreign call-in and call-out mechanisms are clumsy and limited, because of the sizeable gap between the semantics of the two languages, and because of the additional invariants and runtime data required by Scheme. However, since Scheme and C have different and complementary strengths as programming languages, one could wish instead that *hybrid* applications were possible, with independent modules written in different, independently chosen languages.

The "esh" (Embeddable SHell) project at Sun attempts to make just such hybrid applications convenient and efficient, by implementing Scheme as a library rather than an environment, by reducing the invariants and runtime data required by Scheme but not C, and by carefully designing the mechanisms for foreign call-in and call-out to be complete, efficient, and unobtrusive.


1992 ACM LISP & F.P.-6/92/CA
© 1992 ACM 0-89791-483-X/92/0006/0247...$1.50

With the use of esh, modules written in C can supply efficient programmatic primitives, and Scheme modules can supply the control logic which *glues* an application together. This glue can even include a Scheme interpreter usable after the application is delivered, just as [GNUEmacs] supplies a Lisp extension language. Because it is the role of glue that allows it to add unique value to the Unix platform, we have characterized our system as a "shell". But Scheme also makes a reasonable application language in its own right; it seems practical to use esh to deliver applications in which a majority of the source code is Scheme. For example, one internal Sun prototype contains 20k lines of new code, of which more than 95% is Scheme. We do expect this proportion to drop somewhat in the course of product implementation. One Sun product contains about 5k lines of C and 3k lines of esh Scheme.

This paper describes the problems we found bridging the gap between the languages, and how we addressed them.

1. The esh Scheme data formats, run-time system, and packaging were designed to be highly compatible with C and Unix.
2. Scheme supports all C functions and data structures as first-class objects. Based on information obtained from the dynamic linker and from applying an interface compiler to C header files, esh Scheme creates tagged references to arbitrary C values as needed.
3. System interfaces are transliterated from ANSI C into Scheme by a set of rules and conventions designed to express typical ANSI C interfaces concisely and naturally in Scheme's function-oriented style. The conventions also make it easy for a C programmer to understand esh Scheme programs.

## 1.0 Bridging the Gap

The main work in designing the interface between any two languages is reconciling their data models. This is done by putting data and function values into three categories:

[A] peculiar to one language, and not legal at cross-language interfaces

[B] peculiar to one language, but representable in the other

[C] treated as common to both languages

### 1.1 All C Types Have Scheme Classes[1]

The esh implementation of Scheme makes the simplifying assumption that all C data structures are interesting to Scheme. We require that any declarable C function or variable can be passed into and out of a Scheme variable or data structure without loss of information, and preserving type correctness. This goal is consistent with Scheme's use as a glue language.[2] Thus, no C type is in category [A], and as many as possible are in [C].

In fact, every C type gets its own *tag* for use with Scheme's system for runtime dispatch. Every C value is represented, within Scheme, by one tag word, uniquely determined by its C type, and one *value* word. For scalar C values which fit in one word, the value word has a format dictated wholly by C. Other C values (notably doubles) are boxed. Boxed values consume storage managed by the garbage collector. All other C values use only register or stack storage (depending on optimizer vagaries), or are stored as 64-bit double words (of C type `TAG_MEM_T`) inside composite heap objects.

### 1.2 Some Scheme Types are Foreign to C

Not all Scheme tags correspond to C types, although the very fact that esh is implemented in C implies that there is at least one C type which can hold an arbitrary Scheme value. In esh Scheme, `list`, `vector`, and `string` are all abstract types with multiple implementations. None of these objects (except some strings) correspond to any C type; they even signal a runtime type error if you try to pass them to a C varargs function like `printf`. Thus, these types are in category [B] because they can only be represented to C within a the universal two-word box type (`TAG_MEM_T`).

### 1.3 Identifying Common Types

Both Scheme and C have numeric, character, array, and stream types. To maximize interoperability between languages, it is important to treat them as identical where at all possible.

#### 1.3.1 Numbers and Characters

C ints are Scheme integers. (This means our Lisp has 32-bit fixnums.) C chars are Scheme chars. C doubles are Scheme inexact numbers. Other C numeric types are faithfully tagged, and coerce safely among each other, just as they do in C. Scheme arithmetic is generic, as in C, but has additional flexibility due to runtime dispatching.

The Scheme arithmetic operators detect integer overflow, as required by the language. However, unsigned integers, whose semantics are not specified by Scheme, get their arithmetic rules from C, and hence never overflow. Also because of C, Scheme character objects can take part in arithmetic, although `number?` is never true of them.

For the sake of C functions like `fgetc`, the Scheme `EOF` object is tagged as a C character, with the normally impossible value of -1.

#### 1.3.2 Arrays and Vectors

C array values appear to Scheme as vectors. Accesses from Scheme via vector-ref are bounds-checked. An indeterminate array looks like a zero-length vector. Open-ended arrays must be accessed, more clumsily, through unsafe primitives. This limitation has not been a problem in practice since array types per se don't play a large part in C interfaces.

The standard Scheme vector class (produced by make-vector, etc.) is not a C array. This was done for reasons of performance and implementation simplicity: We could have chosen to represent Scheme vectors as C arrays, but this would require vector-producing primitives to retrieve a different tag for each distinct length encountered. Standard Scheme vectors are represented as a count and a pointer to a vector of `TAG_MEM_T` cells. Homogeneous *narrow vectors* in the same format are also supported; they have been useful only occasionally, to conserve storage. At the present time, neither narrow vectors or strings are treated specially by the garbage collector.

---

1. In this paper, the term *abstract type* or just *type* means a set of values, but does not imply committment to a concrete representation. The term *class* or *implementation* refers to a specific, concrete representation for values in some type. In C, classes and types are in 1-1 correspondence, and so we usually say "C type" instead of "implementation". In C++ and Scheme a type can be implemented by any number of classes.

2. We also admit that, whatever Scheme's intrinsic merits, the usefulness of esh in the Unix world depends on its faithfulness to C.

### 1.3.3 Strings and Bignums

The standard Scheme string class (produced by `make-string`, `string-append`, etc.) contains a count and a pointer to a vector of bytes. This allows strings to contain nulls, and allows them occasionally to share bytes with other structures. The C string type is `char*`, which is null terminated rather than counted. This means that the Scheme string type is implemented by at least two classes, one of which cannot be directly processed by C. In essence, the set of values supported by C's string class is a strict subset of all possible Scheme string values.

We handle this embarassing situation by allowing Scheme strings to coerce implicitly to the `char*` class whenever their contents and memory organization allows.[1] Thus, although string constants in Scheme programs are counted arrays, the following program works as expected:

```
(puts "Hello, world.\n")
```

However, if the string constant had contained a null, an exception would be raised at run time when the coercion to `char*` failed. (Varargs does the right thing too: We could also have used `printf` instead of `puts`.)

All strings constructed by Scheme are both counted and null terminated. Our attitude toward C is enthusiastic compromise, rather than capitulation. That is why we insist on using the more powerful counted string type in Scheme, rather than using only `char*`.

It has proven useful to allow even Scheme strings with nulls in them to coerce safely to `void*`, yielding a base address. The idea is that C is going to do something shady with the `void*`, so there's no use adding an integrity check.

A similar situation will arise when multiple precision arithmetic is supported in the future: Integer overflow, which currently leads to an immediate exception, will generate a correct result, represented as a *bignum*. As far as Scheme is concerned, `int` is just a class of the type `integer`, and simple arithmetic is closed for `integer` but not `int`. A bignum result, like a counted array with null in it, will raise an exception later (in an implicit coercion) if Scheme tries to pass the value back to a C function expecting an int.

In effect, most Scheme strings and integers are in category [C], but some are in category [B].

1. This coercion merely produces the base address of the string; there is no copying.

### 1.3.4 Streams

Another type in category [C] is Scheme's `port` type, which is directly represented by C's `FILE*`. This leads to great flexibility intermixing Scheme and C input and output operations. Easy I/O is most important for a glue language. But in order to support Scheme correctly, we had to enhance C's stdio package. Fortunately, this was possible to do (and with binary compatibility!). The Scheme function `char-ready?` required us to implement a C subroutine we called flisten(). We also wanted our Scheme to include the capability to read and write from in-memory strings (the Scheme functions are `open-output-string`, `open-input-string`, and `get-output-string`), so we enhanced the C library again to support this.[2]

This good integration leads to a compromise: The C type `FILE*` is unsafe! This is shown by the following programs:

```
(let ((fp (fopen "file" "r")))
 (fclose fp)
 (fgetc fp))
```

```
{  FILE* fp = fopen("file", "r");
   fclose(fp);
   return fgetc(fp);
   }
```

Our choice to faithfully render C interfaces leads us to faithfully provide access to unsafe features of those same interfaces. (See the section "Keeping Safety" below.) Storage management problems such as the preceding one are a prime source of unsafe interfaces. In the future, we may experiment with a handle-like representation that has an extra level of indirection. Such a handle would be zeroed safely when fclose is called, and in other circumstances would implicitly coerce to `FILE*`.

### 1.3.5 Void

The humble C type (void) is adopted into esh Scheme, and finds new life there. Every Scheme primitive which returns a result undefined by the standard actually returns a C void value. The interactive command loop special-cases void values by printing nothing when a command returns void.

2. The lack of first class in-memory `FILE*` objects in Unix, where other platforms have had this for years, is significant evidence of a tendency toward parochialism in Unix platform vendors, which the Lisp community can have a role in correcting.

#### 1.3.6 Booleans

Since C identifies booleans with integers, and Scheme does not, we could not put these in category [C]. In fact, Scheme booleans do not even coerce to C ints. This has not been a problem in practice. Our header file processor specially treats the type `ix_boolean_t`, nominally a typedef of `int`, to allow C functions to produce and accept Scheme booleans.

This splitting of one C type into multiple Scheme types appears to be a promising technique for dealing with type-deficient interfaces like the XView GUI toolkit, where all object references are of a typedef type that expands to `unsigned`.

### 1.4 Implicit Coercions

As we have already seen, sometimes Scheme objects in category [B] can be coerced into a C type in category [C]. This allows users of Scheme to benefit from specialized non-C value representations, while staying as compatible as possible with C interfaces.

C itself has a moderately complex system of coercions between types. Many are implicitly applied when an actual parameter does not match a formal parameter, or in similar sitations when a local variable is bound. Scheme models these coercions accurately, but on the basis of runtime tag information, rather than statically.

C also has a *cast* construct which can cause unsafe representation punning; Scheme never does this implicitly. Unsafe operations are dealt with below.

Thus, the process of calling a C function from Scheme works like this: If there is a mismatch between the number of actual and formal arguments, signal an error. For each actual, determine the tag representing the C type of the corresponding formal. If the formal calls for a tagged value, just accumulate the tag and value into a new argument list we're building for the C function. If the formal tag is not identical with the actual tag, call a coercion routine to ensure a match or signal an error. Then, accumulate the actual's value field, sans tag, into the C argument list and unbox if necessary. When the new, untagged argument list is complete, call the function on it. When the function returns, box and add a tag as necessary.

The runtime cost of this is essentially linear in the number of tagging and untagging operations. For simple functions, it amounts to a few tens of machine instructions. Many C functions implementing Scheme primitives receive and return fully tagged values. For this extremely simple case, the dispatching overhead is less than ten instructions. Keeping this overhead low is important to system integration, and is enabled by encoding C types as ordinary machine words, rather than bitfields or memory structures requiring a traversal at run time.

### 1.5 Common Functions

Since we represent C functions directly in esh Scheme, it would seem that Scheme functions are in category [C]. Indeed, many important ones are just C functions. For example, `length` takes a tagged value and return an integer. Most Scheme predicates are C functions which return the almost-C type `boolean`.

However, Scheme sometimes requires functions to do things impossible to ordinary C functions. What C type could `list` have, since it takes any number of arguments? The answer is to represent a gathered *restarg* as its own C type (`struct call_args*`), which is never visible to Scheme. (This data structure is organized so as to support efficient stack allocation.) Thus, the C declaration of the function underlying `list` is approximately:

```
extern TAG_MEM_T list_make_list_0v
        (struct call_args* restarg);
```

The schema of tags for C functions is mirrored by another schema for C functions which of similar type, but add the extra restarg. Here's another more complex example:

```
extern tag_t func_static_tag_1v
 (tag_t ret, struct call_args* restarg);
```

This is the function underlying esh Scheme's `+function` constructor, which takes a tag and a restarg (of tags), and returns the tag representing the requested C function type.

There is a third large schema of C function tags, to support C *varargs* functions like `printf`. As mentioned above, esh Scheme handles calls to these by spreading extra arguments on the stack, after coercing each of them to a special C type called "`...`". As a special favor to the many C varargs functions which treat a zero word as an argument terminator, Scheme silently adds a zero every time it builds an argument list for such a function.

#### 1.5.1 Procedure Call Protocol

Scheme's internal function calling protocol is a schema of operations parameterized by number of arguments and presence or absence of a restarg. Each operation has its own methods for all the various classes of Scheme procedure.

The function calling protocol takes restargs into account transparently to Scheme, gathering and spreading restargs as needed, and signalling errors when argument counts cannot be reconciled.

For example, the Scheme code `(list 123 345)` specifies invocation of an operation known as `call_key(2,0)`, where the 0 means "no restarg". The operation is invoked against the function class `func_tag(0,1)`, which represents the C type for `list_make_list_0v` described in the previous section. The mismatch is handled by a default handler for `call_key(2,0)`, which determines that `func_tag(0,1)` will accept a `call_key(0,1)` operation, and gathers the two arguments accordingly.[1]

Optional arguments are handled either via restargs, or by a special procedure class which adjusts the argument list and delegates the call to a corresponding function without optional arguments. In any event, they add no complexity to the procedure call protocol.

Also, statically typed arguments are not taken into account by the call protocol, which assumes that all arguments and return values are fully tagged. Tagging and untagging is the sole responsibility call methods for individual functional tags.

### 1.5.2 Uncommon Functions

As Scheme is a relentlessly function-oriented language, it is not surprising that we found ourselves introducing even more classes of Scheme procedure. For example, lexical closures cannot be represented directly as C functions. (And our optimizing native compiler exercises considerable freedom in its selection of representations for them!) Also, many generic Scheme operations like `car` are in fact low-level dispatchable operations, tagged as such, with call methods that perform the runtime dispatch; this dispatch can be done from C code too, but it is not the same as C function invocation. These additional classes of procedure are all in category [B], since they don't correspond to C functions of any sort.

However, the trick of using implicit coercions, combined with the abstraction naturally afforded by machine-level functions, gives us a very convenient solution: Whenever a category [B] Scheme function is presented as a formal parameter to a C function expecting a functional argument (or is otherwise coerced to a functional C type), it is *boxed* within a C procedure of the required type. The C procedure is created on the heap, and subject to the garbage collector. This boxing is always possible, since all the C procedure needs to do is run a Scheme call operation against the boxed Scheme object.

This convention makes the programming of foreign call-ins extremely simple. In particular, the use of call-ins from GUI toolkits with Scheme-generated callbacks is easy and natural:

```
(define (start-me button
                  client-data call-data)
   ...)

(XtAddCallback start-button XtNselect
               start-me (+void* 0))
```

This example is taken from working X Window System toolkit [Young] code, and occurs within a subroutine whose local variables are fully accessible to `start-me`, and hence to the button widget. A pleasant side effect of this is that client-data values tend become superfluous (and can be initialized to 0, which is what the last argument to `XtAddCallback` does).

During coercion to a C function type, a check is made that the object being boxed is in fact a procedure, but (at present) no other checks are made, regarding argument types and counts. This does not compromise safety, since a call-in from C results in a fully checked call to the object.

Note that coercion of a Scheme function to a C varargs function type always fails, since there is no way for the boxing code to know how many C arguments to retag and pass to the Scheme function.

This section completes the account of how we handle the runtime mechanics of foreign call-in and call-out of procedures. The remaining issues of data structure access and name management will be addressed below.

## 1.6 Types as Values

C types are available in Scheme as first-class values. They are accessible from bindings, either via builtin names such as +int for primitive types, or ix-generated names corresponding to typedefs or structure types. These values are in category [C], because they all have the C type `tag_t`. Derived types can also be created from Scheme library functions. Here is a summary:

| Scheme Expression | C Type |
|---|---|
| `+int` | `int` |
| `+short +char +float +double +void` | *(similarly to int)* |

1. The optimizing compiler handles this reformatting statically.

```
+unsigned +unsigned+short +unsigned+char
                        (unsigned types)
+boolean                ix_boolean_t
(+function +R +A ...)
                        R(A ...)
(+function. +R +A ...)
              R(A ...,
                struct call_args* restarg)
(+function... +R +A ...)
                        R(A ... ...)
(+pointer +T)           T*
+char*                  char*
                        (a convenience)
+void*                  void*
                        (a convenience)
(+array +T N)           T[N]
(+array +T -1)          T[]
(+lvalue +T)            T lvalue
                (C global variables only)
+typedefname            typedefname
+/structname            struct structname
+*                      +TAG_MEM_T
                        (a convenience)
```

The plus signs are merely a naming convention, much like the question marks at the end of Scheme predicate names. These values represent C types, which are implementation-level classes. There are no tag objects which correspond to generic Scheme types like list and vector.

Type values may be used to request type-safe or type-unsafe coercions. Type-safe coercion is requested by using the type name as a unary function (this follows the C syntax, and seems programmer friendly):

```
(+int 1.2) => 1
((+pointer +int) 0)
                => <a null int pointer>
(+void anything) => <a void value>
(+boolean anything-true) => #t
```

This is the same coercion that happens when Scheme calls a statically typed C function. It is internally implemented by a runtime dispatch which takes both the current and desired tags into account.

Type-unsafe coercion is requested using the unsafe `%cast` function, which retags its argument willy-nilly, without touching the argument's value field. This is roughly equivalent in power to the C cast syntax, but is separated cleanly from type-safe coercion:

```
(%cast +int 1.2)
          => <address of a boxed double>
(%cast +int (+float 1.2))
       => <bit pattern of a 32-bit float>
(%cast +boolean 0) => #f
```

The `%cast` function is a necessary evil when dealing with interfaces that are not fully strongly typed, such as XView [Heller]:

```
(define mylabel
   (%cast +char*
          (xv_get mywidget XV_LABEL)))
```

Here is a more complex example, in which a call-in function is described with an inaccurate C type (this happens to be a pre-ANSI interface). In this case, the Scheme function must be coerced to the correct C type (which the interface's header file fails to mention, and must therefore be constructed by hand), and then the result must be casted to the false type expected by the interface:

```
(let ((truetype
          (+function +Notify_value
                     +Notify_client +int)))
  (notify_set_input_func 0
       (%cast +Notify_func
          (truetype input-ready))
       0))
```

## 1.7 Types Peculiar to C

We have discussed category [C] and Scheme types in category [B]. We now list the remaining C types in category [B]. All of them are representable as Scheme objects, although there is in some cases a loss of functionality.

### 1.7.1 Structs and Unions, Enumerations

C structs and unions are represented in Scheme as tagged references, which support field access primitives generated by "ix" as described below. Because Scheme represents them as references, they are not passed by value among Scheme functions the way C functions do. When a C function returns a struct or union to Scheme, it is boxed in a new cell on the heap.

Each `enum` type is a separate Scheme type, which can be coerced to its integer value. Apart from this, Scheme hides the fact that enum values are *really* ints. The coercion from int to enum requires a cast.

#### 1.7.2 Pointers and Lvalues

As with the other C types, the representation of C pointers in Scheme preserves their strong typing.

The Scheme primitives `pointer-ref`, `pointer-`, and `%pointer+` correspond to C `*`, `-`, and + operators. (Note that the last is unsafe.) There are also predicates `pointer?` and `pointer-null?`.

The number 0 coerces to a null of any pointer type. Any pointer coerces to the pointer type `+void*`. The `const` and `volatile` attributes of the pointer's type are ignored. (This is the largest departure we make from the ANSI C.) Unlike in C, a null pointer is not a boolean false.

Finally, a C global variable is treated as an explicit *lvalue* object which can be used with the accessor `lvalue-ref` to fetch the current value of the variable. For the benefit of C functions, lvalues also coerce to their base types with an implicit lvalue-ref:

```
(if (= (lvalue-ref errno) EINTR)
    (try-again)
    (printf "errno = %d!" errno))
```

We did not want to represent C global variables as Scheme global variables, in order to reserve the implementation of Scheme variables per se to the Scheme compiler, rather than tying their representation to that of C.

## 2.0 Invariants Required by Scheme

After reconciling the divergent data models of the two languages, we must also take into account their differing requirements for runtime support. In particular, there are three system invariants required by esh Scheme but not by C:

1. it must be possible for the garbage collector to locate all busy storage
2. it must be possible to perform tail-calls without unbounded stack growth
3. it must not be possible to perform an unchecked unsafe operation

### 2.1 Garbage Collection

Scheme requires automatic storage management, and esh Scheme needs to extend this management to C data structures, since we allow Scheme data structures to contain C values and vice versa. Most implementations of garbage collecting languages give the GC intimate knowledge about the runtime structures of the language. The state of the art in *smart* garbage collectors is surveyed in [Appel]. More recently, work like [Boehm] and [Detlefs] addresses *conservative* garbage collectors with limited knowledge of the system they operate on. The more limited the collector's knowledge, the more conservative it must be in leaving data structures untouched.

The esh garbage collector is extremely ignorant of the data structures it reclaims, and so may be considered *hyperconservative*. It knows the layout of the data areas maintained by `malloc`, the system, and the dynamic linker, but makes no assumptions about where pointers are in the system. It therefore never relocates data. It also deems a heap block to be active if there are any active pointers to any byte within the block. We have yet to find a C program which causes this stodgy thing to free a block erroneously. Its performance is not a bottleneck for the medium-sized applications we use esh in, although occasionally large buffers will cause problems. (The unsafe C function `free` is available for such cases.)

### 2.2 Tail Calls

The following Scheme program should execute forever without blowing the control stack, even if the definitions are in separate modules:

```
(define (even x) (odd (+ x 1)))
(define (odd x) (even (+ x 1)))
(even 0)
```

There is no corresponding requirement on similar C programs, and the existing C control stack and stack frame formats do not take into account the need for such low-overhead *tail-calls* into account.

In esh we use the C control stack for all function invocations. (Even within the interpreter we use the C library routine `alloca` to manage a frame-local stack.) To have defined our own value stack would have made system integration much more difficult, especially with threads coming.

Thus we needed a way for one C function to call another, giving up its stack frame for the use of the callee. This was accomplished with assembly-coded subroutines. The hardest

problem with this stems from the fact that, in most C implementations, a callee does not know how many words of argument storage the caller has allocated, and the caller is responsible for deallocating those words. However, a tail-caller may need to perform this deallocation immediately, and allocate a differing number of argument words, overlaying the storage of the previous argument words. To work around this, we devised an assembly-level protocol for building specially marked, self-sizing, expandable stack frames. If a tail-caller notices that his caller is expandable, he can reformat the stack with confidence. Otherwise, he makes his own frame expandable and performs a regular call.

In the worst case, one out of two stack frames must be expandable; in practice we find only a few turn up. Other tricks keep down the need for expandable stack frames: The interpreter specially recognizes tail calls to interpreted functions, and the native compiler analyzes away many common cases such as loops.

## 2.3 Keeping Safety

One of the principal advantages of Scheme over C is *safety*. This means, roughly, that a Scheme program will not force the machine to do anything that cannot be accounted for within Scheme semantics. In particular, the machine cannot be broken. Compare this to C, where out-of-bounds pointers routinely trash unrelated data structures.

Safety is embodied by a set of system invariants, and a set of checks to ensure those invariants. Like the GC-related invariants, if one module fails to be safe, it can potentially compromise the correctness of all other modules.

When coexisting with C programs, safety must be compromised, if only because a C module is only as safe as the skill and attentiveness of its programmer. More to the point at hand, C modules often expose unsafe interfaces. This can take the form of incomplete typing, as in the case of `printf` and the generic XView accessors. The following esh Scheme programs are unsafe:

```
(printf "%s" n)
(xv_get mybutton XV_LABEL)
```

In such cases, explicit coercions or even casts are needed:

```
(printf "%s" (+char* n))
(%cast +char*
       (xv_get mybutton XV_LABEL))
```

Where possible, it is best to place such code in subroutines which present safe interfaces, so that the problem is localized as much as possible. Here is an abbreviated example from our XView utilities:

```
(define (xv-get-string xvh attr)
 (string-append
   (%cast +char* (xv_get xvh attr))))
```

Another problem with C interfaces is that they expose deallocation operations, so that it is possible to create invalid handles to data structures. In a previous example, this was seen with the C function `fclose`.

Our approach to safety in esh has been to limit breaches of safety to explicit uses of C functions and specially marked unsafe Scheme functions. Unsafe esh Scheme functions, like `%cast`, are all spelled with a leading percent sign. If percent signs are avoided, and C functions are used intelligently, applications will be safe.

This leads to interesting results when designing Scheme primitives that operate on C types. The reader will now see that `pointer-ref` and `pointer-` are safe functions, while `%pointer+` is unsafe. This is because only `%pointer+` can creates questionable pointers. The `pointer-ref` primitive is safe by virtue of the system invariant we chose for pointers: A safe object tagged as a C pointer is either the address of a valid object, or a null pointer. This means that `pointer-ref` need merely perform a runtime check for null before indirecting, in order to maintain safety.

The design described in the section "Strings and Bignums" is also determined by considerations of safety. The relevant invariant is that strings and signed integers do not lose precision when converted to their C representations. This invariant does not apply to the `+unsigned` and `+void*` types, which therefore provide the means for losing precision, when that is desired.

The coercions (as opposed to `%cast` usage) are uniformly safe, in that they do not produce new values which could be used with safe operators to corrupt the system.

Other unsafe primitives create shared and unshared strings and vectors from pre-existing blocks of memory. In that case, the length of the block in question is taken on faith from the programmer.

### 2.3.1 Regarding Type Safety

The correct use of representations is a fundamental and all-encompassing invariant which ensures safety. This invariant is usually called *type correctness*. In languages like C where every type has only one representation, static type checking

is a feasible way of ensuring the correct use of representations. In Scheme, static type checkers are harder to build. Moreover, since Scheme supports multiple representations per type (at least for number and top types), type checking alone cannot ensure correct use of representations. This leads to the use of dynamic dispatch to sort out representations at run time.

In the case where Scheme objects are of C types, the dynamic dispatch merely emulates the same checks an ANSI C compiler would perform statically. The runtime check is guided by the same information that the compiler uses statically: The C type information contained in header files. Thus, esh Scheme loses no C type safety.

## 3.0 Packaging and Naming Issues

We have reconciled the C and Scheme data models sufficiently, and ensured that mixing Scheme and C will not violate system safety, at least in an uncontrollable manner. The final step is to enable Scheme and C modules to be packaged in a form accessible to each other. For C running on Unix, this means that there must be Scheme and header file compilers which can run in batch mode, can be driven from Unix utilities such as "make" and "sh", and turn source files into Unix object modules.

This also means there must be conventions for naming interface parts across module boundaries. We discuss this point first.

### 3.1 Naming Conventions

When a compiled header file is linked or loaded into an esh Scheme application, along with the library implementing the interface, all names defined in the header file are available to Scheme. The Scheme syntax used to access a given interface part depends on the type of that part. Usually this syntax is as close as possible to the syntax used by C clients, allowing for lexical differences between Scheme and C, and allowing for Scheme's dynamic typing.

In order to avoid loss of precision in names, we had to compromise the Scheme standard. The esh Scheme reader normally preserves Unix alphabetic conventions, and does not fold case. C names in Scheme keep their underscores.[1]

For the most part, names from header files are presented in Scheme unchanged. However, since C puts certain names into different namespaces or syntactic categories, while Scheme has a uniform namespace and syntax, it was necessary to add prefixes to certain C names. Here are the rules:

- function
  C functions are presented as Scheme functions of the same name.
- function-like macro
  This works the same as a C function, but requires a hand-written declaration of its effective type.
- enum constant
  This is just a Scheme variable of the same name, bound to the constant's value.
- macro constant
  This works the same as an enum constant (the macro value is evaluated sometime before the first use).
- typedef
  The tag for a typedef name is available by prefixing the C name with the "+" character, as for the builtins like `+int`.
- structure type
  A structure type is obtained from its C struct tag by prefixing the two characters "+/", e.g., `+/_iobuf`.
- structure member
  A structure member `mmm` is accessed using a unary Scheme function of the form `.mmm`. For convenience, there is also the unary Scheme function ->mmm, which applies to struct pointers, and just dereferences the pointer first.
- variable
  A C global variable, such as `errno`, is represented in Scheme as an explicit lvalue object.

### 3.2 Accessors Revisited

An important part of the flavor of C comes from the notion of an *lvalue*, which is a syntactic unit that may be used as either the source or destination of an assignment. This means that one operator, such as pointer indirection, can be used two ways: both to load and store a value.

Standard Scheme uses a different, more verbose idiom. When a data structure is updatable, there is a pair of functions, one for loading and one for storing updatable values. We have extended Scheme with a frequently-seen notion, that of a *settable operation*. This allows us to avoid creating separate functions for getting and setting pointer values and structure elements. All accessor functions mentioned in this paper are settable. All standard Scheme accessors are also settable, and there is an easy way to build your own accessor-based interfaces.

1. This has the useful effect of making C functions stand out slightly from Scheme code.

While a more detailed account of accessors can be found in the comparison with Oaklisp, a single example here will serve to convey the usefulness of accessors. It is a settable version of the function xv-get-string introduced above:

```
(define xv-get-string
 (make-accessor
   (lambda (xvh attr)
    (string-append
      (%cast +char* (xv_get xvh attr))))
   (lambda (value xvh attr)
    (xv_set xvh attr
            (string-append value)))))
```

## 3.3 Interface Extractor

The interface extractor program "ix" scans C header files and stores the information it finds there in Unix object files, which then can be linked with or loaded into Scheme, in precisely the same ways that a compiled C or Scheme module can.

All examples of C interfaces in this paper rely on a previous run of "ix" against a header file. For example, the X Window System toolkit example works only when the proper toolkit intrinsics files have been compiled and linked into Scheme.

We find it useful to batch-compile whole sets of header files at a time, and link them into shared libraries, where the cost of their code and read-only data can be amortized across multiple esh-using client processes. This has done for large modules, such as XView, the X Window System client interface and toolkit, and the standard C libraries.

The header file compiler also works well for small modules. For us, a minimal mixed language application requires not two different compilers, but three. The application requires an ANSI C header file which formalizes the interface between the modules. The C compiler handles both the interface and the C modules, but while Scheme code is compiled by a batch Scheme compiler, the interface is processed separately, by "ix".

## 3.4 Native Compiler

Not much needs to be said of the esh Scheme batch compiler, "eshc", except that it packages its result as a Unix binary file. Scheme load time operations are handled by SVR4 `.init` sections, which contain module-specific code to be run at application startup, or by a corresponding OS-specific feature.[1]

## 3.5 Other Capabilities

### 3.5.1 Block Compilation

As in all Schemes, modularity is enhanced by block compilation. Block compilation can dramatically reduce the storage and CPU time footprint of a Scheme module. We support block compilation via two simple macros and one Scheme extension. The extension allows `define` forms and other forms to be intermingled arbitrarily within any lambda body, with the same meaning as they have at file top-level. The two macros are best illustrated by an example, which should be self-expanatory. This file of code is used during interactive development:

```
;;;; factorial-private.scm:
(define (ifact n acc)
 (if (= n 0) acc
    (ifact (- n 1) (* acc n))))
(define (factorial n)
 (ifact n 1))
(define (safe-factorial n)
 (cond
   ((not (number? n)) (error "NAN!" n))
   ((negative? n) (error "negative!" n))
   (else (factorial n))))
```

The following file of code is compiled for production use:

```
;;;; factorial-public.scm:
;; repackages factorial-private.scm
;; exposes factorial and safe-factorial
(define-block (factorial safe-factorial)
 (include-file "factorial-private.scm"))
```

### 3.5.2 Loading Object Files

The Scheme `load` function operates on Unix shared libaries, and merely dynamically opens the named module, and ensures that the Scheme symbol binder is made aware of it. Due to limitations in Unix, there is no unload or reload functionality.

### 3.5.3 Named Call-in Functions

We support foreign call-in of a named entry point. When generating native code, use of a special form `define-c-callable` will allow that a Scheme function to be bound to a given C name and C type:

```
(define-c-callable (esh_fact n)
      (+function +int +int)
   (let loop (n acc)
```

1. This feature was designed to support C++ static constructors.

```
(if (= n 0) acc
  (loop (- n 1) (* acc n)))))
```

This encourages synergy with developers that are committed to writing exclusively in C or C++.

### 3.5.4 Kernel Packaging

esh Scheme is packaged as a small Unix shared library. It can be linked with a read-eval-print loop module to provide a conventional interactive Scheme system or with other C/ Scheme object files to build arbitrary Unix applications.

### 3.5.5 C-Oriented Scheme Utilities

Certain library functions turned out to be useful. A C++-like function new allows dynamic allocation of arbitrary C objects. The `+typeof` function yields the C type of a C value. The `sizeof` function, applied to a C type, does the expected thing.

There are also utilities for dealing with C arrays, especially argv-style arrays.

## 4.0 Example

The example esh function below prints the names of files found in the directory tree rooted at `dir` whose size, in bytes, is larger than `minsize`. Only files owned by the current user are reported. The function makes use of C types, including some struct types, and a collection of functions from the Unix C library:

`cuserid, getpwnam`

> return the current user id (string) and password entry respectively. The latter is used to lookup the integer user id.

`printf, perror`

> utility functions for printing text and error messages.

`ftw`

> recursively descends a directory tree, breadth first, and applies a callback function to each file and directory name, a pointer the corresponding stat struct, and a flag that roughly characterizes the entry.

This example is interesting because how one can pass an inner function, `eachfile` in this case, directly to a C function. Writing something comparable in C is relatively clumsy because global variables have to be used to pass information from the caller to the callback.

```
(define (find-big-files dir minsize)
 (define uid
   (let ((pwd (getpwnam (cuserid 0))))
    (if (pointer-null? pwd)
      (error "no user id")
      (->pw_uid pwd))))
 (define (eachfile name stat entry-type)
   ;; Verify that this entry is a file,
   ;; it's our file, and it's big
   (if (and
        (= FTW_F entry-type)
        (= uid (->st_uid stat))
        (> (->st_size stat) minsize))
       (printf "%s: %d KBytes\n"
               name (->st_size stat)))
   ;; 0 means continue tree walking
   0)
 ;; start the tree walk
 (if (= -1 (ftw dir eachfile 32))
     (perror "find-big-files failed")))
```

## 5.0 Comparisons with Previous Work

It is useful to compare our problem space and our solutions with those of similar systems.

### 5.1 Common Lisp

All of the commercial implementations of Common Lisp (CL) include support for what is usually called a foreign function interface (FFI) that supports calling functions defined in other languages, usually C and Fortran. [Sexton] reviews the FFI's for Franz ExCL, DEC Vax Lisp, and Lucid Common Lisp.

Our work goes beyond the typical Common Lisp FFI in three ways. First, we support all C types accurately, where the typical FFI handles a small set of scalar types, augmented by a space of memory layout types corresponding to C structs. That is, we allow full foreign call-out. Second, we can dynamically generate call-in functions, which makes all Scheme functions effectively accessible to call-in. Third, our packaging techniques allow Scheme code to run unobtrusively within libraries, even presenting a C interface to other modules.

### 5.2 Oaklisp

The internal architecture of esh Scheme is similar to that of Oaklisp [Pearlmutter]. Both are "object-oriented at the kernel", in that user-visible types routinely have multiple classes representing them.

Oaklisp supports classes in the "classical" Smalltalk [Goldberg] sense: It has an explicit inheritance hierarchy, a uniform object representation system known by the garbage collector, and a metaobject protocol [Kiczales] powerful enough to build a universal data inspector.

Our Scheme makes do without any of these things. We achieve modularization and code sharing "the old fashioned way", with subroutines and abstract data structures, controlling the semantics of runtime dispatch via callbacks into a tiny metaobject protocol. The garbage collector knows only about `malloc` blocks, and within that loose framework we are free to use whatever representations are required for convenience, performance, or consistency with C.

Oaklisp has explicit notions of *setter* and *coercer* which esh Scheme also uses, to better model C coercions and lvalues. There are differences in detail:

| Oaklisp | esh Scheme |
|---|---|
| `((coercer string) x)` | `(+char* x)` |
| `(set! (car x) y)` | same |
| `((setter car) x y)` | `((accessor-setter car) y x)` |

Our coercion syntax is simplified by making tags implement Scheme procedure protocol.

Both systems support settable functions. Both support a generic *setter* operation which retrives the method for setting.

In esh Scheme, as in CLOS, the value precedes the other arguments to the setter. More fundamentally, the semantics of set! are not defined in terms of the setter protocol, but rather the reverse: The setter operation has a default method (written in C) which acts like this:

```
(define ((accessor-setter f) v . a)
   (set! (apply f a) v))
```

The type characterized by `procedure?` has a subtype characterized by `accessor?`. The usual way to make an instance of this type is to call the esh Scheme function `make-accessor` on two functional arguments. The resulting object works like its first procedure argument, by virtue of delegation, not inheritance. Our `accessor-setter` is not a settable operation, and is not even guaranteed to produce equal results for equivalent calls.

In our implementation, this requires a schema of low-level *set-call* operations, parallel to the schema of *call* operations supported by the procedure type. This replication of dispatch operations tends to make `set!` of comparable efficiency to procedure call, an important goal when C data structures are being updated.

The Oaklisp operation `make` is too high-level to correspond to any esh Scheme construct. Also, we treat `set!` of a variable as a primitive, rather than elegantly deriving its value from locatives and setters as Oaklisp does.

### 5.3 Scheme->C

The Scheme-based work most similar to our interface extractor is the declaration compiler called "cdecl", which is a part of DEC's Scheme->C [Bartlett] system. The output of cdecl contains function, constant and type declarations, like that of ix. Like the Common Lisp systems described above, cdecl defines a sub-language for declaring functions and types within Lisp. This language is further processed by the Lisp compiler. The type system it describes is neither Lisp nor C, but a judiciously chosen system of machine-level representations, which overlaps strongly with C. Finally, the runtime representation of Scheme values in this type system uses boxing mechanisms for representing the C values in terms of Lisp types. C functions are boxed in Lisp stub functions, and C pointers are boxed in *foreign pointer* structures.

By contrast, the esh implementation of Scheme represents all C functions and data structures as first-class Scheme objects. This is made possible by a very large space of tags, including one for each ANSI C type. In fact, tags are pointers to type description objects with their own metaobject protocol [Rose], and each tagged value consists of a two words, a tag and an opaque value. Because of this, only multi-word C data need to be boxed on the heap. The output of ix is not a separate Lisp-oriented declaration language, because there is no need for a human-readable specification of interfaces other than ANSI C header files. The output of ix is a Unix object file which contains the runtime structures necessary to represent the ANSI C types, functions, variables, and constants of the desired interfaces.

Because of these tagging techniques, access to C interfaces is very efficient, as efficient as access to *native* Lisp functions and data. C data structures rarely require boxing on the

heap, and functions can be called with minimal tag-checking overhead, usually a few tens of instructions. (Our optimizing compiler will perform these checks statically, if the opportunity arises, generating code equivalent to that of the C compiler.)

Like esh, Scheme->C supports mixed-language applications, and even libraries implemented in Scheme. Like esh, Scheme->C uses the C control stack directly. This limits the ability of both languages to perform some tail calls. In the case of Scheme->C, only inlined functions can be tail-called, while esh gets even cross-module calls right, by the use of some assembly code, if there is no need for reformatting argument lists (apply currently does not perform tail calls). Scheme->C gives `call/cc` continuations indefinite lifetime--a nice trick since the C stack is involved--while esh does not.

### 5.4 ParcPlace Smalltalk

The Objectkit(TM)\Smalltalk C Programming product from ParcPlace Systems is able to assimilate C header files in such a way as to allow C data structures and functions to be accessed from Smalltalk methods. Because this is an automatic process, call-out from Smalltalk is very easy, as with Scheme->C and esh, and unlike the Common Lisp systems.

Since Smalltalk is fundamentally object-oriented, C functions are normally represented as methods on objects. However, since Smalltalk defines a protocol for "closure" objects, C functions can optionally be represented as objects in their own right. For example, this allows a Smalltalk program to pass C functions as arguments to other C functions. As in esh, a Smalltalk closure object can also be passed as an argument to a C function that expects a C function as an argument; Smalltalk constructs the necessary thunk on the C heap automatically, to provide proper calling back into Smalltalk [Deutsch].

## 6.0 Conclusion

The esh implementation of Scheme provides a fresh look at foreign language interfaces, by endeavoring to make C/Unix look as un-foreign as possible, without compromising the C type or value systems.

## 7.0 Acknowledgements

We would like to thank Peter C. Damron, Robert. R. Henry, and Steven Muchnick, for the use of over 1Mb of bibliographic database. Thanks also to Peter Deutsch for up-to-the-minute information on Smalltalk FFIs.

## 8.0 References

[Appel] Andrew W. Appel, "Garbage Collection," ch. 4 of [Lee], 1991.

[Bartlett] Joel Bartlett, "Scheme->C a Portable Scheme-to-C Compiler", Research Report 89/1, DEC Western Research Laboratory, January 1989.

[Boehm] Hans Boehm & Mark Weiser, "Garbage Collection in an Uncooperative Environment," *Software, Practice and Experience*, Sept. 1988.

[Detlefs] David L. Detlefs, "Concurrent Garbage Collection for C++," ch. 5 of [Lee], 1991.

[Deutsch] L. Peter Deutsch, personal communication.

[GNUEmacs] Richard Stallman, *GNU Emacs Manual*, Sixth Edition Version 18, March 1987.

[Goldberg] Adele Goldberg & David Robson, *Smalltalk-80: The Language and its Implementation*, Addison Wesley, 1983.

[Heller] Dan Heller, *XView Programming Manual*, O'Rielly and Associates, 1990.

[Kiczales] Gregor Kiczales, J. de Reveres, and D. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, 1991.

[Lee] Peter Lee, *Advanced Programming Language Implementation*, MIT Press, 1991.

[Pearlmutter] Barak A. Pearlmutter & Kevin J. Lang, "The Implementation of Oaklisp," ch. 8 of [Lee], 1991.

[Rose] John R. Rose, "A Minimal Metaobject Protocol for Dynamic Dispatch," OOPSLA '91 Reflection Workshop.

[Sexton] Harlan Sexton, "Foreign Functions and Common Lisp," Lisp Pointers v. 1, n. 5, March 1988.

[Young] Douglas A. Young, *X Window Systems Programming and Applications with Xt*, Prentice-Hall, 1989.