A Scheme Foreign Function Interface to JavaScript Based on an Infix Extension

Marc-André Bélanger
Université de Montréal
Montréal, Québec, Canada
marc-andre.belanger@umontreal.ca

Marc Feeley
Université de Montréal
Montréal, Québec, Canada
feeley@iro.umontreal.ca

ABSTRACT

This paper presents a JavaScript *Foreign Function Interface* for a Scheme implementation hosted on JavaScript and supporting threads. In order to be as convenient as possible the foreign code is expressed using infix syntax, the type conversions between Scheme and JavaScript are mostly implicit, and calls can both be done from Scheme to JavaScript and the other way around. Our approach takes advantage of JavaScript's dynamic nature and its support for asynchronous functions. This allows concurrent activities to be expressed in a direct style in Scheme using threads. The paper goes over the design and implementation of our approach in the Gambit Scheme system. Examples are given to illustrate its use.

CCS CONCEPTS

• **Software and its engineering** → **Interoperability**; **Compilers**;

KEYWORDS

Foreign function interface, Macros, JavaScript, Scheme

1 INTRODUCTION

In this paper we relate our experience designing, implementing and using a Foreign Function Interface (FFI) in the context of a Scheme implementation hosted on JavaScript. Our system avoids the cumbersome syntax and boilerplate declarations found in typical FFIs and offers a lightweight interface that is both easy to use and expressive.

Cross-language interoperability is a desirable feature of any language implementation. It allows building applications using multiple languages and expressing each part with the most appropriate language. Important factors in the choice of language are the availability of libraries and APIs for the tasks to be done. In a Scheme implementation running in a web browser the support of a JavaScript FFI opens up many interesting possibilities such as accessing the Document Object Model (DOM) and handling events in Scheme code.

ELS'21, May 03–04 2021, Online, Everywhere

https://doi.org/10.5281/zenodo.4711425

FFIs are notoriously implementation-dependent and code using a given FFI is usually not portable. Consequently, the nature of FFI's reflects a particular set of choices made by the language's implementers. This makes FFIs usually more difficult to learn than the base language, imposing implementation constraints to the programmer. In effect, proficiency in a particular FFI is often not a transferable skill.

In general FFIs tightly couple the underlying low level data representation to the higher level interface provided to the programmer. This is especially true of FFIs for statically typed languages such as C, where to construct the proper interface code the FFI must know the type of all data passed to and from the functions. As a simple example, here is a program using Gambit Scheme's C FFI[9] to interface to the C library's `ldexp(`x`,`y`)` function computing $x \times 2^y$:

```
(c-declare "#include <math.h>") ;; get ldexp C prototype
(define ldexp (c-lambda (double int) double "ldexp"))
(println (ldexp 10.0 -3)) ;; prints 1.25
```

Note the use of type specifiers in the `c-lambda` to indicate the type of the arguments (`double` and `int`) and result (`double`). The FFI defines some mapping between the Scheme data and the C types, and raises an error for incompatible types (e.g. the Gambit C FFI will raise an error when a C `double` is expected and a value other than a Scheme inexact real is used). There is even wider variation in how different FFIs handle more complex constructs like variadic functions, higher order functions, multiple return values, pointers, arrays, structures, classes, continuations and threads (a notorious hard case is interfacing to the C library's `qsort` function which uses universal pointers and a callback). FFIs are also difficult to use from the REPL and interpreted code, if at all possible.

There is an opportunity to simplify the interfacing code when the host language is a dynamically typed language that supports dynamic code evaluation (i.e. `eval`). By interfacing through *expressions* rather than the *function* level, we can leverage a Scheme reader extended with infix notation support to generate and evaluate host language expressions. Our work proposes a new FFI design based on those ideas for interfacing Scheme and JavaScript that is easy to use in the common case yet also supports more complex use cases including asynchonous execution and threads. Our design enables *natural* cross-language programming that programmers can pick up quickly. We first go over our design and its implementation, followed by an exposition of potential uses.

2 SYNTAX AS INTERFACE

Programmers using an FFI are expected to be knowledgeable in both the foreign and native languages; in our case JavaScript and Scheme respectively. We use the term *native* as the opposite of *foreign* and **not** to mean machine code.

Some Lisp/Scheme FFIs cast the foreign constructs to a parenthesized syntax to make them usable within the native language. As an illustrative example, JScheme[1] interfaces to Java using functions whose names follow the "Java Dot Notation" containing special characters: `(.method obj ...)` for calls to methods, `(Constructor. ...)` for calls to constructors, `Class.field$` for static field members, etc. For example: `(.println System.out$ "hello")` performs the same operation as the Java code `System.out.println("hello")`.

This decoupling of syntax and semantics can be confusing to the programmer who must essentially write semantically foreign code but with a more or less contrived mapping of foreign constructs to the native syntax (e.g. there are 9 different but similar looking Java Dot Notation rules in JScheme). This adds an intellectual burden and prevents *cut-and-paste* of existing foreign code snippets and idioms into native code. Our point of view is that the foreign language's syntax should be preserved as much as possible in native code in order to express naturally the foreign constructs which don't map well to the native language, such as method calls and accessing member fields. Moreover, the difference in syntax helps distinguish which of the two languages is being used at various points in the program. This works particularly well in languages such as Scheme and JavaScript that have easily distinguishable syntaxes: prefix and infix notation.

2.1 Scheme Infix eXtension

In our FFI, infix expressions within Scheme code are considered to be foreign code. A single backslash must precede the infix expression to *escape* temporarily from the prefix syntax. As a simple example the following native code will print the current day of the week twice to the JavaScript console:

```
(define date \Date().slice(0, 15))
(define day (substring date 0 3))
\console.log(`day)
\console.log(`(substring \Date().slice(0, 15) 0 3))
```

The first line calls out to JavaScript to retrieve the date as a Scheme string (stripped of the time of day using the JavaScript `slice` method). The second line extracts the day of the week with a Scheme call to `substring`. The third line reads the Scheme variable `day` and uses the JavaScript `console.log` method to print it out. The last line is similar but with an inline Scheme call to `substring`. Note the use of a backquote to switch back to the prefix syntax temporarily within the infix form. The expressions marked with a backquote are evaluated in the Scheme environment whereas the rest of the infix form is evaluated in the JavaScript global environment. This is why the identifiers `Date` and `console` refer to JavaScript globals and the identifiers `date`, `day`, and `substring` refer to Scheme variables. As shown in the last line it is possible to nest prefix and infix forms.

The FFI's implementation is simplified by Gambit's existing reader which supports reader macros and which has a default setup to invoke the Scheme Infix eXtension (SIX) parser when a backslash is encountered. After a complete infix form is read the reader continues parsing using the prefix syntax. Note that the Gambit reader does not use `\` to escape symbols as in Common Lisp and some other Scheme implementations. Vertical bars are the only supported symbol escaping syntax.

Similarly to other reader macros, the SIX parser constructs an s-expression representation of the infix form's AST. This representation can easily be viewed by quoting the infix form and pretty printing it, for example `(pp '\console.log(`day))` prints:

```
(six.infix
 (six.call
  (six.dot (six.identifier console)
           (six.identifier log))
  (quasiquote day)))
```

The system achieves the JavaScript FFI semantics by defining a `six.infix` macro, as explained in the next section.

The grammar supported by the SIX parser is given in the Gambit user manual and the details are mostly uninteresting here. Two aspects are nevertheless noteworthy.

First of all the grammar combines syntactic forms from multiple languages including C, JavaScript, Python and Prolog (without implementing any of those grammars fully) and has a few extensions, such as `` `X `` to parse `` `X `` using the prefix syntax. The choice of using `` ` `` to switch back to prefix is motivated by the fact it is seldom used in the grammars of infix based languages. It is also evocative of Scheme's quasiquote notation, but with `\` and `` ` `` in place of `` ` `` and `,`.

The infix operators have the precedence and associativity of the JavaScript language. The SIX parser was originally designed to be used in an undergraduate course on compilers to easily construct ASTs. Even though the parser supports multiple syntaxes, it is problematic to force the parser to restrict the syntax to a subset, or to extend the grammar itself, as this introduces a phasing problem. It would require introducing a read-time mechanism (such as Racket's `#lang` feature[10]) to select the grammar, which is something we want to avoid so that a source file can combine code for multiple host languages (possibly guarded by a `cond-expand` dependent on the compilation target).

Secondly, supporting infix forms within prefix forms required a few syntactic concessions related to identifier syntax and whitespace handling. Whitespace is significant when it is outside of infix form parentheses and braces. The expression `(list \1+2 \3+4-5)` evaluates to `(3 2)` whereas `(list \(1 + 2) \3+4 -5)` evaluates to `(3 7 -5)`. This is important to keep the syntax lightweight. In an earlier version of the SIX parser whitespace was not significant and infix forms were required to end with a semicolon, but this was less visually pleasing, so the current parser makes the semicolons optional outside of infix braces.

The use of a Scheme identifier immediately after a ` can cause issues because Scheme allows many more characters in a symbol than in JavaScript. Consequently the SIX parser will have the surprising behaviour of treating `\`x-f(y)` as an expression containing a reference to the identifier `x-f`. The programmer can circumvent this issue by using `\(`x - f(y))`, `\(`x)-f(y)`, or `\`|x|-f(y)` that unambiguously reference `x`.

2.2 The `six.infix` Macro

As with any FFI there is a need to bridge the semantic gap between the native and foreign languages. An important aspect is the conversion of values between the languages so that a value generated in the Scheme code gets converted to the appropriate value in JavaScript. Conversion from JavaScript values to Scheme values is also needed for returning results back to Scheme. Conversions in both directions are also needed to allow JavaScript code to call Scheme procedures. Scheme procedures and JavaScript functions are themselves considered to be data with a mapping to the other language.

The FFI must also bridge the semantic gap of the control features of the languages. Specifically, Gambit Scheme supports proper tail calls, first class continuations, and threading (SRFI-18 and SRFI-21) thanks to a green thread scheduler implemented in Scheme using first class continuations. JavaScript does not support these features. However it does offer asynchronous functions which are similar to threading in that they allow concurrent activities. The details of the mapping between languages is explained in Section 4.

The FFI's semantics are implemented using a suitable definition of the macro `six.infix`. This macro traverses the SIX AST of the expression and extracts all the Scheme expressions wrapped in a `quasiquote`. These are the *Scheme parameters* of the expression. These Scheme parameters are given names by generating JavaScript identifiers (`___1`, `___2`, ...). The SIX AST gets translated to a string of JavaScript code representing an asynchronous function whose parameters are these identifiers and where the body of the function computes the expression using these identifiers. As an example, the SIX expression `\(`s).repeat(10)`, that can be used to repeat a Scheme string `s` 10 times, leads to the creation of this JavaScript function:

```
async function (___1) { return ___1.repeat(10); }
```

Using the standard data mapping this JavaScript function will be converted to a Scheme procedure *p* and `\(`s).repeat(10)` expands to code that performs the call `(`*p* `s)`.

There is a concern with this approach related to phasing. The `six.infix` macro is expected to work in compiled code and also through the REPL and Scheme's `eval`. This is solved by a JavaScript `eval` of the function definition to create dynamically the JavaScript function and the corresponding Scheme procedure. This dynamic evaluation is required because at Scheme compilation time the JavaScript environment has not yet started running (for example the Scheme program could be compiled on a desktop computer ahead of time).

Scheme	**JavaScript**
`#!void`	`undefined`
`()`	`null`
`#f`/`#t`	`false`/`true`
fixnum, e.g. `12`	number, e.g. `12`
flonum, e.g. `1.2`	`_Flonum`
bignum, e.g. `999999999`	`_Bignum`
ratnum, e.g. `1/2`	`_Ratnum`
cpxnum, e.g. `1+2i`	`_Cpxnum`
character, e.g. `#\a`	`_Char`
pair, e.g. `(1 . 2)`	`_Pair`
string, e.g. `"abc"`	`_ScmString`
symbol, e.g. `abc`	`_ScmSymbol`
keyword, e.g. `abc:`	`_ScmKeyword`
structure, port, table, ...	`_Structure`
vector, e.g. `#(1 2)`	`Array`, e.g. `[1,2]`
u8vector, e.g. `#u8(1 2)`	`_U8Vector`
... other homogeneous vectors	
f64vector, e.g. `#f64(1.2 3.4)`	`_F64Vector`
procedure of n parameters	parameterless function

Figure 1: GVM's representation of the Scheme types in JavaScript

To avoid a call to the JavaScript `eval` at every evaluation of a given `six.infix` call site, the expansion uses a simple caching mechanism to remember the Scheme procedure obtained during the first evaluation. A Scheme box is used as a cache. It initially contains a string of the JavaScript function definition, and on the first execution it is mutated to contain the result of calling the JavaScript `eval` on that string and converting the result to a Scheme procedure. This is done by the procedure `##host-function-memoized` which takes the box as its sole parameter and returns the Scheme procedure. So to be precise, `\(`s).repeat(10)` expands to:

```
((##host-function-memoized
  '#&"async function (___1)
      { return ___1.repeat(10); }")
 s)
```

with `##host-function-memoized` defined as:

```
(define (##host-function-memoized descr)
  (let ((x (unbox descr)))
    (if (string? x)
        (let ((host-fn (##host-eval-dynamic x)))
          (set-box! descr host-fn)
          host-fn)
        x)))
```

3 GAMBIT VIRTUAL MACHINE

Before discussing in more detail the implementation of the FFI it is important to briefly go over the Gambit Virtual Machine (GVM) which is the Gambit compiler's intermediate language. Thanks to this abstraction it is possible to retarget the compiler with moderate effort, and indeed there are backends for several languages both high-level (C, JavaScript, Python, ...) and machine languages (x86, arm, riscv, ...). In

the case we are concerned with here, the compiler translates GVM instructions to JavaScript.

To support the GVM each Scheme type is mapped to a corresponding representation as a JavaScript type. The mapping for some types is direct when a JavaScript type supports the same operations. This is the case of `#f`, `#t`, `()`, `#!void`, vectors, and fixnums which are mapped to `false`, `true`, `null`, `undefined`, `Array`, and numbers respectively. It is important for performance to use JavaScript numbers as a representation of fixnums so that operations on small integers can be done without an extra layer of boxing/unboxing.

Figure 1 gives a list of the Scheme types and their representation in JavaScript. All the JavaScript classes supporting the GVM are prefixed with `_`. In most cases a class is used to group the information related to a type, for example a `_Pair` contains the `car`/`cdr` fields of a Scheme pair and a `_Char` contains the Unicode code of a Scheme character. Scheme strings are not mapped to JavaScript strings because those are immutable. Instead a `_ScmString` contains an `Array` of Unicode codes. Scheme symbols are not mapped directly to the (relatively new) JavaScript `Symbol` type because the GVM stores a hash code and other information in symbols.

Scheme procedures are mapped to parameterless JavaScript functions. What is peculiar about this mapping is that the JavaScript function is used to represent a control point in the code, similarly to a code label in assembly language. These control point functions take no parameters and return a new control point function or `null`. Jumping from one point in the code to another is the job of the *trampoline* which is the loop `while (pc) pc = pc();` where `pc` is the current control point function. A control point function can jump to a new control point by returning this new control point to the trampoline which will transfer control to it. This approach is needed to support tail calls properly and also to perform strict checking of the parameter count (JavaScript does not check or report parameter count mismatches). When Scheme code calls a procedure the parameters, a parameter count and a *return address* (another control point function) will be stored in registers and the stack of the GVM (JavaScript global variables) before returning the procedure's control point function to the trampoline to jump to it.

Control point functions, the trampoline and an explicit representation of the stack are the basic elements needed to implement closures, continuations and threads similarly to implementing these types in machine language. The thread type is a structure with several fields, one of which is the continuation of the thread. When a thread needs to be suspended, its current continuation is captured and stored in the thread structure so that it can be invoked later when the thread's execution needs to resume. Mutexes and condition variables are structures which contain a queue of threads, which are the threads blocked on them. A thread scheduler implemented in Scheme keeps track of a queue of runnable threads and moves threads out of this queue when the threads block on a mutex or condition variable. The scheduler is preemptive, forcing the current runnable thread to the end of the runnable thread queue when it has been the currently running thread for more than a small time interval (typically 0.01 second).

Scheme		**JavaScript**
`#!void`	⟷	`undefined`
`#f`/`#t`	⟷	`false`/`true`
fixnum, flonum	⟷	number, e.g. `12`, `1.2`
bignum, e.g. `999999999`	⟶	number, e.g. `999999999`
ratnum, e.g. `1/2`	⟶	number, e.g. `0.5`
character, e.g. `#\a`	⟶	number, e.g. `97`
exact integer, e.g. `42`	⟵	`BigInt`, e.g. `42n`
string, e.g. `"abc"`	⟷	string, e.g. `"abc"`
symbol, e.g. `abc`	⟶	string, e.g. `"abc"`
keyword, e.g. `abc:`	⟶	string, e.g. `"abc"`
vector, e.g. `#(1 2)`	⟷	`Array`, e.g. `[1,2]`
`()`	⟶	`Array`, e.g. `[]`
pair, e.g. `(1 . 2)`	⟶	`Array`, e.g. `[1,2]`
pair, e.g. `(1 2 3)`	⟶	`Array`, e.g. `[1,2,3]`
table	⟶	object, e.g. `{a:1,b:2}`
u8vector	⟷	`Uint8array`
... other homogeneous vectors		
f64vector	⟷	`Float64Array`
procedure of n parameters	⟷	function of n parameters

Figure 2: FFI mapping of types between Scheme an JavaScript

4 FFI MAPPING OF TYPES

The FFI defines the mapping of types between the native and foreign languages. The mapping is designed to be convenient and intuitive to allow commonly used values to be be mapped to the other language to what is expected by a programmer, and be consistent. The conversions need not have a link with the GVM's mapping of Scheme to JavaScript types, which was chosen to achieve good execution speed of pure Scheme code. The mapping is given in Figure 2.

4.1 Simple Types

The conversion functions which implement this mapping, the JavaScript functions `_scm2host` and `_host2scm`, are called when there is an inter-language call when converting the parameters and the result. It is desirable for values to be invariant when they are sent to an identity function in the other language (i.e. that the round-trip does not change the value in the sense of `equal?`). However this is not possible for all values. The Scheme values `#!void`, `#f`, `#t`, strings and homogeneous vectors are bidirectionally mapped to the JavaScript values `undefined`, `false`, `true`, strings and typed arrays respectively, so they have ideal round-trip behaviour. Scheme vectors are bidirectionally mapped to JavaScript `Array`s, however the elements of the array need to be recursively converted. So the round-trip behaviour of vectors/`Array`s will depend on the round-trip behaviour of their elements.

Numbers need to be mapped carefully because JavaScript has two numerical types, *number* and `BigInt`, that correspond to Scheme's inexact reals and exact integers respectively. However, they are not consistently used that way in typical

code (for example JavaScript arrays are almost never indexed with `BigInt` which is a fairly recent addition to the language). For that reason it is more convenient for Scheme exact integers to be mapped to JavaScript numbers. When a JavaScript number is converted to Scheme, it will become a fixnum value if it has an integer value falling in the fixnum range, otherwise (if it has a fractional part or is outside the fixnum range) it becomes a flonum value. When a JavaScript `BigInt` is converted to Scheme, it will become an exact integer (either a fixnum or bignum depending on its value). Scheme bignums and rationals are also mapped to numbers. Scheme characters are mapped to the number that is their Unicode code.

Scheme symbols and keywords are converted to JavaScript strings. Scheme pairs and lists are converted to JavaScript `Array`s with recursively converted elements.

4.2 Procedures

Scheme procedures are mapped bidirectionally to JavaScript functions and they accept the same number of parameters. In the conversion from one language to the other, calls to the appropriate conversion functions are added to convert the parameters and the result. In other words, when a Scheme procedure p is converted to the JavaScript function f, a call of f in JavaScript must pass JavaScript values that will be converted to the corresponding Scheme value for processing by p. When p delivers its Scheme result it will be converted to JavaScript and returned for the call to f. The situation is similar for a JavaScript function that is converted to Scheme.

Asynchronous functions and the `Promise` type were added to JavaScript to avoid the deeply nested Continuation Passing Style (CPS), aka. "callback hell", that commonly occurs when using CPS to perform asynchronous processing. In a language with threads, such as Gambit Scheme, asynchronous processing can instead be expressed in a direct style using threads that wait for the availability of the next piece of data or event. Our FFI implements a mapping of JavaScript promises and asynchronous functions to Scheme threads, making asynchronous processing easier to use.

It is important to realize that, due to the presence of threads, Scheme procedures may take an arbitrary long time to complete if they block the current thread on an I/O operation or mutex or condition variable until some event unblocks the thread and allows the procedure to return. So for a smooth integration with the JavaScript execution model, Scheme procedures must be mapped to JavaScript asynchronous functions. Similarly, a JavaScript asynchronous function may take an arbitrarily long time to deliver a result, so if Scheme code calls a JavaScript asynchronous function it may cause the current Scheme thread to effectively block. However, this must not happen deep inside JavaScript code because in that case the Scheme thread scheduler itself would be unable to continue scheduling runnable threads (in effect the scheduler itself would be blocked).

This is solved by using the *Promise API* and a JavaScript to Scheme callback that notifies the Scheme thread scheduler when a promise is *settled* (either *fulfilled* with a value or *rejected* with an exception). An asynchronous JavaScript function f is converted to a Scheme procedure p that ends with a call to the `##scm2host-call-return` procedure that receives the promise result of the asynchronous function. The Scheme thread must wait for the promise to be settled. This is achieved with a mutex that is initially in a locked state and that the Scheme thread tries to lock (thus blocking at that point). When the promise is settled a Scheme callback is called which stores the result (in the mutex *specific* field) and unlocks the mutex, allowing the Scheme thread to determine if the result is normal or an error. The following code shows how this synchronization is implemented:

```
// JavaScript side

function _when_settled(promise, callback) {

  function onFulfilled(value) {
    // call the Scheme callback asynchronously
    _async_call(false, false, // no result needed
                callback,
                [_host2scm([value])]);
  }

  function onRejected(reason) {
    // call the Scheme callback asynchronously
    _async_call(false, false, // no result needed
                callback,
                [_host2scm(reason.toString())]);
  }

  promise.then(onFulfilled, onRejected);
}

;; Scheme side

(define (##scm2host-call-return promise)
  (let ((mut (make-mutex)))

    ;; Setup mutex in locked state
    (mutex-lock! mut)

    ;; Add callback for when promise is settled
    (when-settled ;; defined in JS as above
     promise
     (scheme ;; pass-through (see next section)
      (lambda (result) ;; callback
        (mutex-specific-set! mut result)
        ;; wake up waiting Scheme thread
        (mutex-unlock! mut))))

    ;; Wait for promise to be settled
    (mutex-lock! mut)
    (let ((msg (mutex-specific mut)))
      (if (vector? msg)      ;; Promise was:
          (vector-ref msg 0) ;; fulfilled
          (error msg)))))    ;; rejected
```

Non asynchronous JavaScript functions can be encountered by `_host2scm` in a variety of situations, including converting data structures containing functions and global functions such as `alert` and `fetch`. The above code is a slight simplification of the actual code which must also handle calling a non asynchronous JavaScript function which (typically) does not return a promise. This is done by dynamically testing the type of `##scm2host-call-return`'s parameter to determine if it is a promise.

Because the handling of SIX expressions creates a definition of a JavaScript asynchronous function, a call to that function always returns a promise. The expansion of the `six.infix` macro will contain a Scheme call of the JavaScript asynchronous function converted to Scheme. Consequently the Scheme thread will implicitly wait for the asynchronous JavaScript processing to complete before continuing. This decouples the control flow of the Scheme thread scheduler and the JavaScript task queue, allowing other Scheme threads to run while the asynchronous call is executing.

A similar decoupling is necessary for Scheme procedures that are converted to JavaScript asynchronous functions. When called, the JavaScript function creates a promise and an `Array` packaging the Scheme procedure to call, the parameters and a JavaScript callback, and calls `_async_call` to add this `Array` to a *callback queue*. A dedicated *callback loop* Scheme thread reads this queue, performs the corresponding Scheme call and settles the promise accordingly (fulfilled or rejected depending on whether a Scheme exception was raised) by calling the JavaScript callback.

4.3 Pass-Through Types

In some cases it is not desirable for values to be converted implicitly according to the previously described rules. An important case is when a value created by one language needs to be stored by the other language for passing back to the originating language unchanged at a later time. For this purpose the FFI defines two *pass-through* types which are treated specially by the conversion functions, represented by the `_Scheme` and `_Foreign` JavaScript types. These types simply box a Scheme and JavaScript value respectively. In Scheme a `_Scheme` value is constructed with the procedure call `(scheme` *`val`*`)`. In JavaScript a `_Foreign` value is constructed with the function call `foreign(`*`val`*`)`.

The `_scm2host` conversion function acts as the identity function when passed a `_Scheme` value. Similarly the `_host2scm` conversion function acts as the identity function when passed a `_Foreign` value. However, when passed a `_Foreign`, the `_scm2host` conversion function unboxes the value to get back the JavaScript value originally passed in the call `foreign(`*`val`*`)`. Similarly, when passed a `_Scheme`, the `_host2scm` conversion function unboxes the value to get back the Scheme value originally passed in the call `(scheme` *`val`*`)`. With these rules it is possible for the programmer to achieve ideal round-trip behaviour (in the `eq?` sense) for any value by inserting explicit calls to `scheme` and `foreign` when the normal conversion must be disabled.

The `foreign` function can also be used to bypass the implicit promise synchronization. If the programmer wants the calling Scheme thread to continue execution without waiting for the asynchronous call to complete then a `Promise` object can be returned to Scheme by wrapping it in a call to `foreign`. Waiting for a promise `p` to be settled is as simple as writing `` \`p `` as shown in the following example:

```
;; define a JS function that takes time to complete

\sleep=function (ms) {
        return new Promise(function (resolve) {
                                setTimeout(resolve,ms);
                               });
      }

(define p \foreign(sleep(5000)))  ;; does not wait

\sleep(1000)  ;; pause Scheme execution for 1 sec

\`p  ;; waits for the remaining part of 5 secs
```

The `scheme` procedure can also be used to write JavaScript code that directly accesses the GVM's value representation. This can be useful to implement special operations or special purpose conversions of Scheme values. A `_Scheme` value has a `scmobj` field that contains the Scheme object (more precisely its GVM representation using JavaScript objects). For example `` \`(scheme "abc").scmobj.codes[1] `` evaluates to 98, which is the Unicode code of the second character of the Scheme string `"abc"`.

For convenience, any Scheme object not mentionned in Figure 2 is converted to a `_Scheme` value. Similarly, any JavaScript object not mentionned is converted to a `_Foreign` value. As a consequence, a data conversion between languages always succeeds. For example a Scheme complex number will be converted to a `_Scheme` value, allowing the GVM representation to be accessed using JavaScript, as shown in the following code:

```
(define num 1+2i)
(println \(`num).scmobj.real)  ;; prints 1
\(`num).scmobj.imag=9          ;; mutate object
(println num)                  ;; prints 1+9i
```

This shows that by accessing its GVM representation the complex number can be mutated even though it is a Scheme constant. This clearly exposes implementation details to the programmer, which is a double edged sword (useful in some contexts but dangerous if not used properly). A programmer should mainly rely on the FFI mapping shown in Figure 2 and seldom if ever use the GVM object representation details given in Figure 1 that is more likely to change in future versions of Gambit or when special compiler options are used (indeed the Gambit compiler's `compactness` setting may cause the use of shorter names for the fields of GVM objects).

5 EXAMPLES

We can now show various examples that illustrate the qualities of our design. We begin with a trivial *hello world* program and move on to more involved use cases. We hope these examples convincingly show the simplicity of use and terseness of the code interfacing Scheme and JavaScript. The reader may want to try the examples on the online Gambit Scheme REPL at `https://gambitscheme.org/try/` using a *cut-and-paste* of the code shown.

5.1 Interfacing with the DOM

One of the most obvious use cases for the FFI is interfacing with the DOM. The browser effectively acts as a graphical user interface for Scheme, just as it would for JavaScript. As a first example, let's consider inserting a DOM node in the page to render some text, as shown in Figure 3.

```
(define msg "<h1>Hello!</h1>")
(define top "afterbegin")

\document.body.insertAdjacentHTML(`top, `msg)
```

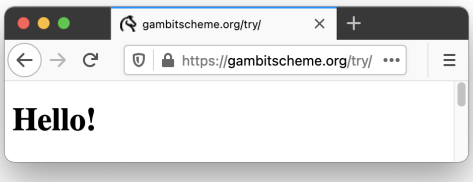

Figure 3: Simple modification of the DOM.

This defines the HTML code to insert and then calls the `insertAdjacentHTML` method of the *body* element of the page. Note that this directly invokes the JavaScript DOM API without writing Scheme wrappers. This code should feel natural and be self-explanatory for any programmer knowledgeable in JavaScript and Scheme, a stated goal of our design. The example in Figure 3 could have been written as

```
(define body \document.body)  ;; <-- foreign object
\(`body).insertAdjacentHTML(`top, `msg)
```

where the *body* element is stored as a foreign object in Scheme. Such use still feels natural and allows for modularization in more involved code, such as when developing a library.

5.2 Event Handling

Event handlers and listeners constitute the foundation of interactive browser user interfaces. It is easy to register a Scheme procedure as a callback to an event listener. Because of the implicit mapping of Scheme procedures to JavaScript functions, any Scheme procedure can be used as a callback to process DOM events triggered on the page. The following code will track the mouse movements and log the x, y coordinates to the console as the mouse is moved.

```
(define (handler evt)
  (update \(`evt).clientX \(`evt).clientY))

(define (update x y)
  \console.log(`(object->string (list x: x y: y))))

\document.addEventListener("mousemove", `handler)
```

Figure 4: Registering a Scheme procedure as an event listener callback.

The `handler` event listener callback passes the *mousemove* event's *clientX* and *clientY* coordinates to the `update` procedure. The latter simply logs the coordinates to the console using JavaScript's `console.log`.

5.3 Interfacing with Libraries

Modern web apps typically make use of multiple external JavaScript libraries. Our FFI allows Scheme code to easily interface to such libraries. JQuery is a widely used library that facilitates interacting with the DOM. Figure 5 is a representative example.

```
(define html
  (string-append
    "<button>Toggle visibility</button>"
    "<p class='first visible'>First paragraph</p>"
    "<p class='second hidden' "
    "style='display: none'>Second paragraph</p>"))

\document.body.insertAdjacentHTML("beforeend", `html)

(define (toggle evt)
  (let ((hidden \$("p.hidden"))
        (visible \$("p.visible")))
    \((`hidden).removeClass("hidden")
               .addClass("visible")
               .toggle())
    \((`visible).removeClass("visible")
                .addClass("hidden")
                .toggle())))

\$("button").click(`toggle)
```

Figure 5: Interfacing Scheme with JQuery through the FFI.

After defining and inserting the HTML, the `toggle` event handler is defined and assigned to the *click* event using JQuery. The event handler uses JQuery's `$` function to find the elements corresponding to a *selector*. In the example, the selectors find every `<p>` element with class *hidden* or *visible*. The `toggle` handler uses the JQuery `removeClass`, `addClass` and `toggle` methods to hide or show the element in question. An element with class *hidden* will see its class change from *hidden* to *visible*, and its actual visibility toggled by JQuery's `toggle` method, and vice versa.

The SIX expressions in the body of the handler's `let` are wrapped in parentheses to allow writing a multi-line

expression which is very similar to the conventional style used in JavaScript. The pattern of selecting and mutating DOM elements is very common and forms the basis of rich user interfaces and web applications and is clearly easily achieved with our FFI design.

5.4 Asynchronous Updates

Asynchronous processing is a useful approach to decouple the UI and application logic. In this example we use JavaScript's `fetch` API to get resources from other web servers, specifically the weather reports of New-York and Miami. The program in Figure 6 uses the JavaScript `fetch` asynchronous function to request a JSON formatted weather report from the server `forecast.weather.gov` . The temperature is shown for each city and is updated every 10 seconds.

The updating is handled for each city by creating one Scheme thread per city. Each thread loops on the operations that fetch the JSON weather report, transfers the temperature to the DOM, and sleeps for 10 seconds. Note that the Scheme code hides from view inside the `fetch-json` procedure the promises and asynchronous functions that are operating at the JavaScript level.

```
(define (fetch-json url)
  \fetch(`url).then(function (r) { return r.json(); }))

(define (url loc)
  (string-append
   "https://forecast.weather.gov/MapClick.php?"
   "lat=" (cadr loc) "&lon=" (caddr loc)
   "&FcstType=json"))

(define (html loc)
  (string-append "<h3><span id='" (car loc)
                 "'>?</span> F -- " (car loc) "</h3>"))

(define (show-weather loc)
  \(document.body
     .insertAdjacentHTML("beforeend", `(html loc)))
  (let ((elem \document.getElementById(`(car loc))))
    (thread (lambda ()
              (update-weather elem loc 10)))))

(define (update-weather elem loc period)
  (let loop ()
    (let ((json (fetch-json (url loc))))
      \(`elem).innerText=`(temperature json)
      (thread-sleep! period)
      (loop))))

(define (temperature json)
  \(`json).currentobservation.Temp)

(for-each show-weather
          '(("New-York" "40.78333" "-73.96667")
            ("Miami"    "25.76000" "-80.21219")))
```

Figure 6: Asynchronously updating weather reports using threads.

5.5 Parallelism

Figure 7 is our last example. It shows how the use of threads for asynchronous processing can improve performance. The program starts off by defining the `future` and `touch` forms of Multilisp[11] to easily express parallelism. They are the basis of the `pmap` procedure which is like `map` but processes all elements concurrently. The rest of the code uses `pmap` to fetch 43 images[1] asynchronously and adds them to the web page. This program is an order of magnitude faster than one using plain `map` because it takes advantage of the inherent external parallelism in the web servers and network.

```
(define-syntax future
  (lambda (stx)
    (syntax-case stx ()
      ((future expr)
       #'(thread (lambda () expr))))))

(define touch thread-join!)

(define (pmap f lst)   ;; "parallel" map
  (map touch (map (lambda (x) (future (f x))) lst)))

(define memo
  (string-append
   "Scheme_-_An_interpreter_for_extended_"
   "lambda_calculus.djvu"))

(define (page n)
  (string-append
   "https://upload.wikimedia.org/wikipedia"
   "/commons/thumb/1/1e/" memo
   "/page" (number->string n) "-593px-" memo ".jpg"))

(define (fetch-blob url)
  \fetch(`url).then(function (r) { return r.blob(); }))

(define (->URL blob)
  \URL.createObjectURL(`blob))

(define (show url)
  \document.body.insertAdjacentHTML(
   "beforeend",
   "<img src='"+(`url)+"' width=200px>"))

(define images
  (pmap (lambda (n) (->URL (fetch-blob (page n))))
        (iota 43 1)))

(for-each show images)
```

Figure 7: Downloading a set of images in parallel.

6 RELATED WORK AND CONCLUSION

C FFIs are offered by Scheme implementations such as Racket[5], Chez Scheme[8], Larceny[12], Bigloo[14] and Gambit Scheme[9]. These essentially propose a domain-specific

[1]The pages of the original Scheme report!

language (DSL) to facilitate interfacing through foreign function declarations, something we wish to avoid.

FFIs to dynamically typed languages exist in languages and software such as Hop[15], Haskell[7], Kotlin[2], PharoJS[6], Pyodide[4] or Racket[5]. Of these, Haskell's *Foreign Expression Language*, PharoJS and Racket's facilities fall into the DSL category. Hop, Pyodide and Kotlin allow seemingly more natural access to JavaScript code. This is facilitated by Kotlin and Python's syntactic similarity to JavaScript. However, these methods are in essence either like writing Python to a string and passing it to Python's `eval` in the case of Pyodide (which is essentially CPython compiled to WebAssembly), or evocative of the C FFI function declarations in the case of Kotlin. JScheme[1] and LIPS[3] offer yet another way of interfacing with JavaScript by leveraging a *dot notation*, wherein Java or JavaScript semantics is mapped to Scheme through syntactic convention.

Hop's ability to syntactically distinguish computations that should occur on the server or the client resembles our escaping mechanism to switch between languages. This is reminiscent of quotation/antiquotation in SML[16], which allows to splice host-language expressions in foreign code. The `C (*Tick C*)[13] language also offers a mechanism using a backquote for escaping between languages which is reminiscent of our own. Racket provides facilities for modifying its reader and expander which can be used to read and execute custom languages by using the `#lang` form[10]. These features, while certainly powerful, are more complex than our solution, yet share the quality of allowing a programmer to switch back and forth between languages.

All things considered, our work distinguishes itself from other FFIs most clearly by its use of a Scheme reader extended with an infix notation parser. This allows our FFI to interface host and foreign languages at the expression level, enabling a more concise and natural style. The FFI's ability to interface JavaScript asynchronous functions with Scheme threads transparently also simplifies combining Scheme programs with asynchronous JavaScript code and libraries.

ACKNOWLEDGMENTS

This work was supported by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

[1] JScheme Reference Manual. Retrieved March 15, 2021 from http://jscheme.sourceforge.net/jscheme/doc/refman.html.
[2] Use JavaScript code from Kotlin | Kotlin. Retrieved March 15, 2021 from https://kotlinlang.org/docs/js-interop.html.
[3] LIPS: Powerful Scheme based lisp interpreter in JavaScript. Retrieved March 15, 2021 from https://lips.js.org/.
[4] Pyodide – Version 0.17.0. Retrieved April 22, 2021 from https://pyodide.org/en/0.17.0/.
[5] Eli Barzilay and Dmitry Orlovsky. Foreign interface for PLT Scheme. In *Proceedings of the Fifth ACM SIGPLAN Workshop on Scheme and Functional Programming*, pages 63–74, Snowbird, Utah, 2004.
[6] Noury Bouraqadi and Dave Mason. Mocks, Proxies, and Transpilation as Development Strategies for Web Development. In *Proceedings of the 11th edition of the International Workshop on Smalltalk Technologies*, pages 1–6, Prague Czech Republic, August 2016. ACM. https://dl.acm.org/doi/10.1145/2991041.2991051.
[7] Atze Dijkstra, Jurriën Stutterheim, Alessandro Vermeulen, and S. Doaitse Swierstra. Building JavaScript Applications with Haskell. In Ralf Hinze, editor, *Implementation and Application of Functional Languages*, volume 8241 of *Lecture Notes in Computer Science*, pages 37–52, Berlin, Heidelberg, 2013. Springer. https://doi.org/10.1007/978-3-642-41582-1_3.
[8] R. Kent Dybvig. Chez Scheme Version 8 User's Guide. *Cadence Research Systems*, 2009. Retrieved March 15, 2021 from https://www.scheme.com/csug8/.
[9] Marc Feeley. Gambit v4.9.3 manual, 2019. Retrieved on March 15, 2021 from http://www.iro.umontreal.ca/~gambit/doc/gambit.pdf.
[10] Matthew Flatt, Robert Bruce Findler, and PLT. The Racket Guide. Retrieved April 22, 2021 from https://docs.racket-lang.org/guide/index.html.
[11] Robert H. Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, October 1985. https://doi.org/10.1145/4472.4478.
[12] Felix S. Klock II. The Layers of Larceny's Foreign Function Interface. In *Workshop on Scheme and Functional Programming*, Vancouver, British Columbia, 2008.
[13] Massimiliano Poletto, Wilson C. Hsieh, Dawson R. Engler, and M. Frans Kaashoek. 'C and tcc: a language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems*, 21(2):324–369, March 1999. https://doi.org/10.1145/316686.316697.
[14] Manuel Serrano. Bigloo, a Practical Scheme Compiler, March 2021. Retrieved March 15, 2021 from http://www-sop.inria.fr/indes/fp/Bigloo/.
[15] Manuel Serrano, Erick Gallesio, and Florian Loitsch. Hop: a Language for Programming the Web 2.0. In *Proceedings of the First Dynamic Languages Symposium*, pages 975–985, Portland, Oregon, 2006.
[16] Konrad Slind. Object language embedding in Standard ML of New Jersey. In *Proceedings of the Second ML Workshop*, CMU SCS Technical Report, Pittsburgh, Pennsylvania, 1991. Carnegie Mellon University.