An R7RS Compatible Module System for Termite Scheme

Frédéric Hamel
Université de Montréal
Montréal, Québec, Canada
frederic.hamel@umontreal.ca

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

ABSTRACT

The Termite Scheme language is an existing extension of Gambit Scheme that has features well suited for programming heterogeneous distributed systems using a message passing style. The language supports sending messages containing procedures and continuations, which simplifies migrating tasks between nodes during their execution.

A longstanding issue with the original implementation of Termite is that compiled procedures and continuations can only be sent to other nodes if the compiled code is already loaded in the program receiving the message. This is tedious to arrange in the typical case, and hard or impossible for hot code updates which are an important use case (updating a service without interrupting its execution).

Our work has implemented a solution to this problem: an R7RS compatible module system that automates the distribution of compiled code. The module system uses a version control system to manage module versions and provide a way to distribute code from network accessible repositories. Modules are identified uniquely using the repository location and version number. This allows multiple versions of the same module to coexist in a program, an essential feature to support hot code updates.

We explain the implementation of our module system and how it solves various issues related to Termite Scheme and programming distributed systems. Through an experimental evaluation we have observed speed improvements for RPC of close to one order of magnitude.

CCS CONCEPTS

• **Software and its engineering** → **Modules / packages**; **Modules / packages**; **Distributed systems organizing principles**;

KEYWORDS

Distributed systems, Concurrency, Remote Procedure Call, Mobile code, Modules, Scheme

ACM Reference Format:
Frédéric Hamel and Marc Feeley. 2020. An R7RS Compatible Module System for Termite Scheme. In *Proceedings of the 13th Europeen Lisp Symposium (ELS'20).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.5281/zenodo.3742443

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ELS'20, April 27 28, 2020, Zürich, Switzerland
© 2020 Copyright held by the owner/author(s).
https://doi.org/10.5281/zenodo.3742443

1 INTRODUCTION

Distributed systems consist of a set of interconnected computational nodes. Nodes interact by sending and receiving messages from other nodes over a communication network. Each node may have a special purpose or there can be more or less duplication of their function. The World Wide Web is a notable example that is good to keep in mind to understand some of the issues. It has server and client nodes, they typically don't run the same server and client programs, and the nodes are not centrally managed.

The implementation of a distributed system consists of developing the programs installed on the nodes that perform the coordination of the nodes' actions with those of other nodes. In a sense, the set of the nodes' programs constitutes a global program that must be correct. The challenging development issues we consider in this paper are the following:

- **RPC:** How is a remote procedure call (RPC) implemented when the program in the sending and receiving nodes haven't been designed together?
- **Code update:** How is a node's program updated when a bug is fixed or a better version is available?
- **Task migration:** How is a service moved to a new node when the underlying platform must be changed (operating system update, hardware upgrades, reboot, ...)?
- **Continuous operation:** How are service interruptions avoided in the above situations?

The Termite Scheme language [19] has been designed as an extension of Gambit Scheme [14] to simplify programming distributed systems and provide solutions to these issues. It borrows concepts from the Erlang programming language [4], but using Scheme syntax and semantics. A particularly interesting feature in our context, not found in Erlang, is the ability to send messages containing continuations.

The Gambit system on top of which Termite is implemented offers many features useful to program distributed systems. It generates portable C code that can be compiled and run on any OS and architecture (32/64 bits, little/big endian, ...). The serialization and deserialization of objects are independent of their machine representation, allowing the transmission of most objects between nodes of a distributed system with different architectures. In particular, procedures and continuations can be serialized. Moreover interpreted and compiled code can be freely mixed in the same program. The serializable procedures allow using a higher-order programming style across nodes, which is useful for implementing RPC. The serializable continuations allow capturing the state of a process and sending it to another node to resume it there, which is useful for code update and task migration.

Unfortunately, the original implementation of Termite has shortcomings when serializing closures and continuations. When the receiving node has no knowledge of the code it receives, it must run the code interpreted which is typically much slower than if it was compiled. In this case, essentially the source code file's AST is serialized, making messages larger. Moreover, this large structure will be sent again if another instance of the closure is sent. When the code is compiled, the messages are compact, but the code must be available in compiled form on the receiving node. This requires a tedious and error-prone setting up of the nodes' programs that would be unsustainable in the context of large independently evolving distributed systems, such as the Web, and difficult to use for RPC, code update and task migration.

Our work aims to allow sending code between nodes that both run native compiled executables. This is not a simple task considering nodes may have different operating systems and architectures, such as ARM, i386 or x86_64, so compiled procedures cannot, in general, be sent as machine code.

To reach this goal our approach delegates to the receiving node the compilation of the modules. Code can be transmitted between machines of different architectures without the usage of cross-compilers, which can be challenging to setup robustly. The code will work on all platforms supported by Gambit such as Linux, macOS, Windows, etc.

An essential property of the module system is that modules must have a globally unique name that includes their version. This allows the coexistence of multiple versions of modules in a program, a situation occuring when a node is updated with an improved version of its code without interruption. Our approach benefits from the use of a version control system to manage the versions of modules in a disciplined way.

The implementation of code migration has been done in different programming languages such as Java [18], JavaScript [22], Tcl [20], Erlang [16, 21, 23], and Scheme [3, 9, 13, 17]. Our work distinguishes itself from these efforts in allowing compiled code to be migrated transparently between nodes regardless of their architecture and operating system, and without the destination node having prior knowledge of the code.

Section 2 is a brief tutorial of the Termite Scheme language. Section 3 explains existing features of Gambit used in the implementation of our module system, which is the subject of Section 4. In Section 5 we evaluate the performance experimentally. Finally, Section 6 discusses related work.

2 TERMITE SCHEME LANGUAGE

Termite [19] applications consist of multiple Termite `node`s exchanging data in a message-passing style similar to Erlang [15]. A `node` is an abstraction of a computing device that is distinct from the physical nodes (machines) of the distributed system. In practice a `node` corresponds to an operating system process and the physical nodes of the distributed system may contain a single or multiple Termite `node`s.

Within each `node` there are multiple running threads. Threads are uniquely identified across the distributed system with a `upid` that indicates its location (i.e. a `node` and a sequence number within that `node`).

Each `node` is identified with an IP address and port. The procedure `make-node` is the constructor of `node` identifiers. The `node-init` procedure starts the `node`'s TCP server and registers built-in services (*spawner*, *linker*, *publisher*, etc.) The `node`'s TCP server allows clients to connect to it remotely. Without this the `node` would not be visible on the network.

Services in Termite are nothing more than threads receiving messages in a loop, performing pattern matching on them and executing actions according to the result of the matching. A service is created with the `spawn` procedure which takes a thunk to be executed and an optional local name for the service and returns a thread object. Each thread has a mailbox that buffers the messages it receives. By default services created with `spawn` are only visible to the threads in the current `node`. A service can be published globally (to other nodes) by the procedure `publish-service` that registers the thread object under a specific name in a dictionary within the *publisher* service. That service resolves services by name.

Communication between `node`s is performed with the following procedures:

- `(! dest msg)`
- `(? [timeout [default]])`
- `(?? pred? [timeout [default]])`
- `(!? dest msg [timeout [default]])`

The procedure `!` sends the message *`msg`* to the mailbox of the *`dest`* thread. The procedure `?` waits for a message to appear in the mailbox and retrieves it, with the optional parameter *`timeout`* specifying how long to wait before returning the *`default`* value. If no default value is given and the timeout expires, an error is raised. The procedure `??` is similar to `?` but filters the received message using the predicate *`pred?`*. For the common "send request then receive response" communication pattern there is the `!?` procedure that is a composition of `!` and `?`, which also automatically adds a unique tag to the messages to match the response with the request.

A message can also be received with the `recv` form which pattern matches the message. This form is typically used to dispatch the messages in the implementation of a service.

The following example shows a typical use of these forms to implement a local service computing the square of a number and a client requesting to square 5 (both in the same `node`):

```
(define square-server
  (spawn
    (lambda ()
      (let loop ()
        (recv
          ((from tag 'square x)  ;; message pattern
           (! from (list tag (* x x))))
          (msg
           (warning "Ignored message " msg)))
        (loop)))))

(!? square-server (list 'square 5)) ;; => 25
```

```
;; ping.scm (running on node1)
(declare (block))
(import (termite))

(define pong-server
  (remote-service 'pong-server node2))

(define new-server
  (spawn
    (lambda ()
      (let loop () ;; code that will be migrated
        (recv
          ((from tag 'clone)
           (call/cc
             (lambda (k)
               (! from (list tag k)))))
          ((from tag 'ping)
           (! from (list tag 'pong)))
          (('update k)
           (k #t)))
        (loop)))))

(node-init node1)

(!? pong-server 'ping) ; => gnop
(! pong-server (list 'update (!? new-server 'clone)))
(!? pong-server 'ping) ; => pong
```

```
;; buggy-pong.scm (running on node2)
(declare (block))
(import (termite))

(define pong-server
  (spawn
    (lambda ()
      (let loop ()
        (recv
          ((from tag 'ping)
           (! from (list tag 'gnop))) ;; BUG!
          (('update k)
           (k #t)))
        (loop)))))

(node-init node2)

;; publish the pong server
(publish-service 'pong-server pong-server)
```

Figure 1: Hot code update example

Here the created thread loops forever receiving lists of the form (*from tag* `square` *x*) where *from* is the source thread, *tag* is a unique tag created for this request/response, and *x* is the number to square. The thread responds with a message of the form (*tag result*) where *result* is the square of *x*.

2.1 Hot code update

In many distributed system implementations, updating the code of a node requires restarting the program running on that node. Performing hot code updates while a program is running is useful to change its behaviour without interrupting the service. This can be to fix a bug or to extend the service. In Termite, this is possible in part because both procedures and continuations are serializable.

A basic ping-pong example is enough to unveil issues of the original implementation of Termite. In this example there are two `node`s each running a thread; one that sends `ping` and the other that replies `pong`. The Termite Scheme code in Figure 1 demonstrates how to perform a hot code update of the server without interrupting its service. The actors in this scenario are the server (`buggy-pong.scm`) and the client (`ping.scm`). Note that the server's implementation has a deliberately introduced bug: it replies to a `ping` request with the message `gnop` instead of `pong`.

The client application designed to fix the server is composed of a thread that contains the new behaviour of the server. The thread must handle the same messages as the buggy pong server to be compatible. Additionally, it is distinguished in two ways, first it fixes the response message to `pong` when receiving `ping`, second it handles the message `clone` which captures the continuation of the client thread which will be sent to the buggy server to fix it.

The client starts by creating a local service with `spawn`. It pings the buggy pong server and prints the (incorrect) result, creates and sends the continuation of the `new-server` to the buggy pong server in an `update` message. Then, it re-pings the server and prints the (correct) result. In the original implementation of Termite this works correctly when the code is run interpreted, but it fails when compiled. This is because the message sent by the client contains a continuation that refers to return points in compiled code that do not exist on the server that receives the message. Our module system offers a mechanism solving this problem.

2.2 RPC/RMI

Remote procedure call (RPC) and remote method invocation (RMI) are both mechanisms that allow remote execution of code on a remote computer. The essential difference is that RMI is object-oriented while RPC is not. Java RMI supports direct transfer of serialized Java classes and distributed garbage collection. A remote call[5] can be described as the following sequence of events:

(1) The client calls a local stub with parameters passed to it in a normal way.

(2) The client stub packs the parameters into a message. This is called marshalling.
(3) The client sends the message to a server on the remote node.
(4) The server stub unpacks the parameters of the message. This is called unmarshalling.
(5) Finally, the server stub invokes the procedure with the arguments. The result is marshalled then sent back to the client.

In Termite Scheme and Erlang, RPC servers can be implemented by creating a service that dispatches the messages to the right procedure. The square service example given earlier is a RPC server allowing a single procedure to be called. In general, the message dispatch used in the server constrains the procedures that can be executed to the ones handled by the dispatcher.

Termite has the `on` procedure to circumvent this constraint by allowing the execution of a thunk on any `node`. This procedure takes as parameters a `node` and a thunk and returns the result of calling the thunk on that `node`, for example `(on node2 (lambda () (directory-files)))`. This simplifies the implementation of RPC servers to a simple `node` initialized with the `node-init` procedure.

The procedure `remote-spawn` is similar to `spawn` but the thread it creates is on the `node` specified as a parameter. This thread can then be used to execute specific code. No interface code is required because the client explicitly sends the thunk to the destination `node`.

The power of these features rests on the transparent unrestricted code migration mechanism possible with our module system.

3 EXISTING GAMBIT FEATURES

The implementation of our module system uses the following existing Gambit features.

3.1 Symbolic paths

When a filesystem path begins with `~~name` it expands to the path bound to that name in the *symbolic path dictionary*. This extension to the filesystem path syntax is convenient for accessing directories whose location depends on the system configuration or command line arguments. For example `~~lib` is bound to the directory containing the builtin Gambit libraries and `~~userlib` is bound to the directory containing user installed libraries.

3.2 Module loader

A source code file may contain the declaration `(##supply-module module-id)` to identify the file as the module `module-id`. It can also contain a set of `(##demand-module module-id)` forms indicating dependencies on functionality provided by the modules identified by `module-id`. When the file is compiled, these properties are embedded in the generated code and available to the module loader. When the compiled file M is loaded the Gambit runtime system will ensure that M's required modules are loaded first. The required modules are searched using the *module-id* in a set of directories known as the module search order, which by default is `~~userlib` followed by `~~lib`.

3.3 Object serialization

The serialization of objects in Gambit is done with the procedure `object->u8vector` which takes as parameters the object to serialize and optionally a *transform* procedure which is called on every sub-object inside the object in order to customize the serialization process. The result is a `u8vector` (vector of bytes).

The serialization of most objects is straightforward. A first byte indicates in the upper bits the type of the object, and possibly some basic property such as its length in the remaining bits of the byte (if it fits otherwise in the following bytes). This is followed by the serialization of each field of the object. For example the vector `#(1 2 3)` is serialized to the four bytes `#x23 #x51 #x52 #x53`. The first byte indicates the vector type and a length of 3, and the remaining bytes the type and value of the small integers 1, 2 and 3.

Circular references and shared objects are handled by keeping track of the position of serialized objects in the byte stream and using a special type that refers to a previously serialized object by its position in the stream.

The machine independent serialization of compiled procedures, closures and continuations is based on the serialization of control points. There are control points for procedure entry points, closure entry points and non-tail call return points.

The serialization of control points is the key to serialize closures and continuations. Gambit uses a flat closure representation [7]. A closure is a vector-like object containing the free variables and a reference to a control point (the closure's entry point). Gambit uses the *incremental stack/heap strategy* [10] for managing continuations. A continuation is a chain of continuation frames, either stored on the stack or the heap (the details of the representation are given in [13]).

Continuation frames, like closures, are vector-like objects containing values and a reference to a control point (the return point of a non-tail procedure call). These objects are serialized similarly to vectors. Each of their fields needs to be serialized, the only particularity is that one of the fields is a control point.

For historical reasons Gambit uses the term *subprocedure* for control points. Each subprocedure has a *parent* which is the toplevel procedure that contains it. The subprocedures contained in a given parent are assigned a machine independent integer index identifying that control point in the parent: its *id*. Each toplevel procedure has itself as a parent and an id of 0. These attributes can be obtained with the procedures `(##subprocedure-parent subproc)` and `(##subprocedure-id subproc)` which access meta-information maintained by the runtime system. The inverse operation, namely the retrieval of the subprocedure with a given *parent* and *id*, is performed by the procedure

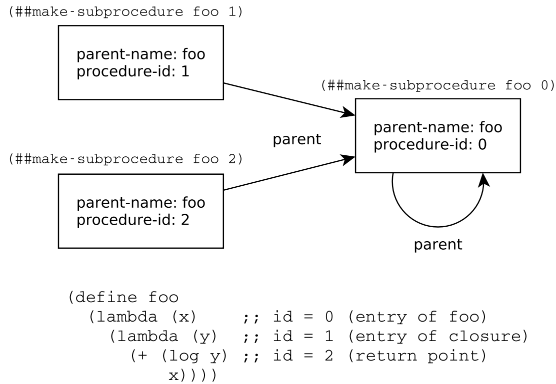

Figure 2: Machine independent control point identification

`(##make-subprocedure` *`parent id`*`)`. Figure 2 shows a procedure containing three control points, their logical relationship, and how they can be retrieved with `##make-subprocedure`.

The last issue to address is the machine independent identification of the parent. Gambit has a block compilation declaration, i.e. `(##declare (block))`, that tells the compiler that all global variables defined in the file are not mutated in other files. When this declaration is used, the Gambit compiler assigns a name to each toplevel procedure, which is typically the name of the global variable used in the toplevel `define`, or a name derived from the filename when the lambda-expression is not used in a toplevel `define`. The serialization of parent procedures uses this name, which is obtained with the procedure `(##subprocedure-parent-name` *`subproc`*`)`. Deserialization needs to recover the reference to the parent given its name, and this is done with the procedure `(##global-var-primitive-ref` *`name`*`)`. This procedure fails when a toplevel procedure with that name does not exist in the receiving node's program. Our module system catches this case to trigger the dynamic loading of the missing compiled code with a mechanism explained in Section 4.1. The main point is that the name of the toplevel procedure contains enough information to find the corresponding source code, compile it and load it.

3.4 Namespaces

To avoid clashes of global variables and toplevel macros between modules, Gambit partitions names into namespaces. A name is *qualified* when it contains a `#`, such as `math#pi`, and is *unqualified* when it does not, such as `sqrt`. This notation is inspired by Curtis' *et al* module system for Scheme [11]. A namespace is a prefix that is added to unqualified variable and macro names to make the name qualified. For example, the `math#` namespace applied to the unqualified `sqrt` results in the qualified name `math#sqrt`. A namespace is either the special empty namespace or a name that ends with a `#`.

```
(##namespace ("math#")
             ("" (def define) if < * -))

(def (fact n)
  (if (< n 2) 1 (* n (fact (- n 1)))))
```

Figure 3: Namespace declaration example

Gambit's `##namespace` declaration controls which namespace is added to unqualified names in the scope of the declaration (which is the rest of the file if at toplevel). Three forms of this declaration exist (to simplify we have used the `ns#` namespace):

(1) `(##namespace ("ns#"))`
(2) `(##namespace ("ns#"` *`name1`*`...))`
(3) `(##namespace ("ns#" (`*`name1 alias1`*`)...))`

In the first form, all unqualified identifiers in the scope of the declaration will be augmented with the `ns#` prefix. In the second form only the unqualified names *`name1`*... are augmented. In the third form the unqualified names *`name1`*... are renamed to *`alias1`*... and then the `ns#` prefix is added.

A toplevel `##namespace` declaration placed at the beginning of a file can map the names used in the rest of the code to appropriate namespaces. Figure 3 shows a sample use to implement a small math library. The declaration maps `def` to `define` (in the empty namespace), and everything else except `if`, `<`, `*`, `-` to the `math#` namespace. Consequently this code actually defines the `math#fact` procedure. If the code contained other definitions they too would define names in the `math#` namespace. Another module could access this procedure directly with the qualified name `math#fact`, or with `fact` if in the scope of a `(##namespace ("math#" fact))`, or with `factorial` if in the scope of a `(##namespace ("math#" (factorial fact)))`.

4 OUR MODULE SYSTEM

4.1 `define-library` form

To help with the adoption of our module system we have designed it to be compatible with the R7RS standard [24]. The main form for defining libraries (which is synonymous to modules in this paper) is the `define-library` form which has the following syntax:

`(define-library` *<library name>*
<library declaration> `...)`

The first argument is the library name; a non-empty list of identifiers such as `(scheme base)`. The name is followed by library declarations which can be any of the following forms, of which the last six are extensions to the R7RS syntax offering fine control over the compilation and linking process:

- `(export` *<export spec>*`...)`
- `(import` *<import set>*`...)`
- `(begin` *<command or definition>*`...)`
- `(include` *<filename>*`...)`
- `(include-ci` *<filename>*`...)`
- `(include-library-declarations` *<filename>*`...)`

```
(define-library (github.com/fred hello)
  (export hi)
  (import (only (scheme base) define)
          (rename (scheme write) (display show)))
  (begin
    (define (hi str)
      (show "hello ")
      (show str)
      (show "\n"))))
```

Figure 4: The library `(github.com/fred hello)` exporting the procedure `hi`

```
(define-library (gitlab.com/zoo cats)
  (import (only (scheme base) define)
          (github.com/fred hello @1.0))
  (begin
    (define (main)
      (hi "lion")
      (hi "tiger"))))
```

Figure 5: The library `(gitlab.com/zoo cats)` which depends on version `1.0` of the library `(github.com/fred hello)`

- (cond-expand *<cond expand features>*...)
- (namespace *<namespace>*): e.g. (namespace "X11#")
- (cc-options *<options>*...): e.g. (cc-options "-O3")
- (ld-options *<options>*...): e.g. (ld-options "-lX")
- (ld-options-prelude *<options>*...)
- (pkg-config *<options>*...): e.g. (pkg-config "X11")
- (pkg-config-path *<path>*...)

The library's variable and macro definitions are typically contained in the `begin` declaration. It is also possible to put the definitions in another file that is included with one of the `include` forms. The `cond-expand` form allows conditional activation of the library declarations it contains depending on the system's support for specific features.

The `export` declaration indicates the list of variables and macros defined by the library that are accessible to code importing this library. Like with R7RS the `export` declaration can indicate that specific (internal) names are renamed to other (external) names.

A library declares a dependency to another library with the `import` declaration. This declaration identifies the imported library. The `import` declaration may restrict the subset of exported names that are imported (by default all exported names are imported). The exported names may also be renamed. This is specified in the *<import set>* with the forms `(only ...)`, `(except ...)`, `(rename ...)`, and `(prefix ...)`. We have extended the R7RS syntax for imported library names to allow a trailing `@version` that indicates the specific version required. Any mobile code libraries imported must have a version indicator for reliable operation of the module system.

Figures 4 and 5 are an example of libraries defined with our extended `define-library`. Figure 4 implements a library that exports the procedure `hi`. It uses an `import` declaration that imports only the name `define` exported from the standard library `(scheme base)` and all the names exported from the standard library `(scheme write)`, but with `display` renamed to `show`. Figure 5 is a library that depends on version `1.0` of the library defined in Figure 4.

The R7RS specifies that the library name "is used to identify the library uniquely when importing from other programs or libraries". In the context of mobile code libraries, our system has the stronger requirement that libraries are identified uniquely across all nodes of the distributed system. This is achieved by requiring mobile code libraries to be in code repositories hosted by version control system servers (which could be a public service such as `github.com` or a privately managed server) and to use the location of the repository in the name of the library. The use of a version control system allows multiple versions of the library to be stored in a single repository; each identified with a specific commit tag. The use of a network accessible repository makes it possible to obtain the code from any node of the distributed system. In our example the library name `(github.com/fred hello)` encodes the location of the repository, i.e. `http://github.com/fred/hello`.

The module system uses the library name and version to construct a unique library identifier. The version is either implicit (the current commit tag of the version control system) or indicated explicitly in the `import` declaration. All identifiers in the library name are concatenated with a `/` separator followed by an `@` and the version identifier (implicit or explicit). A trailing `#` is added to get the unique namespace for the library. Unless the library has a `namespace` library declaration to force the namespace (which is mainly useful for builtin libraries), the namespace derived from the library name is used to construct the qualified names of the (toplevel) variables and macros defined by the library. Due to the guaranteed namespace uniqueness, name clashes between libraries are not possible, including different versions of the same library. In our example, if the version of the library is `1.0`, the definition of `hi` is in reality defining the global variable with the qualified name `github.com/fred/hello@1.0#hi` whereas if the version is `1.5` it is `github.com/fred/hello@1.5#hi` that is defined.

Having the library location and version information in the global variable name provides valuable information to the subprocedure deserialization mechanism. When the procedure `(##global-var-primitive-ref name)` fails because a procedure with that name does not exist on the receiving node the system uses the procedure name to determine where to fetch a copy of the repository for the library containing that procedure, and which version of the repository is needed. A local copy of the repository at the required version is then made and the Gambit compiler is invoked to create the compiled code which is dynamically loaded into the running

```
;; expansion of (define-library (github.com/fred hello) ...)

(##declare (block))

(##supply-module github.com/fred/hello@1.0)

(##namespace ("github.com/fred/hello@1.0#")
             ("" define
                 (show display)
                 write-shared
                 write
                 write-simple))

(define (hi str)   ;; defines github.com/fred/hello@1.0#hi
  (show "hello ") ;; calls display
  (show str)       ;; same
  (show "\n"))     ;; same


;; expansion of (define-library (gitlab.com/zoo cats) ...)

(##declare (block))

(##supply-module gitlab.com/zoo/cats@2.0)
(##demand-module github.com/fred/hello@1.0)

(##namespace ("gitlab.com/zoo/cats@2.0#")
             ("" define)
             ("github.com/fred/hello@1.0#" hi))

(define (main)  ;; defines gitlab.com/zoo/cats@2.0#main
  (hi "lion")   ;; calls github.com/fred/hello@1.0#hi
  (hi "tiger")) ;; same
```

Figure 6: Expansion of version `1.0` of the library (`github.com/fred hello`) and version `2.0` of the library (`gitlab.com/zoo cats`)

program, allowing the deserialization to resume. The compiled code is kept locally to avoid costly recompilations if that version of the library is used again in the future. The local copy of the repository is also kept to avoid fetching it again if another version of the library is required.

4.2 Implementation of `define-library`

The `define-library` form is implemented as a macro that expands into existing Gambit forms. For the libraries shown in Figures 4 and 5, expanding the `define-library` forms produces the code shown in Figure 6, for versions `1.0` and `2.0` of the libraries respectively.

The expanded code starts with a `(##declare (block))` declaration that informs the compiler that variables defined in the library will not be mutated in other libraries (this semantics is part of the R7RS specification). This enables some optimizations by the compiler, such as constant propagation and inlining, and it also causes the compiler to assign to each toplevel procedure a name that includes the library's location and version, necessary for the deserialization process.

This is followed by a `(##supply-module` *`library-id`*`)` that provides the module loader with the identity of the library and version implemented by the code. For each imported library (with a non-empty set of definitions), a `(##demand-module` *`library-id`*`)` is generated to inform the module loader that the specified library must be loaded first. The handling of exported and imported names is done through a generated `##namespace` form that maps the names used in the library's code to the qualified names. The definition of all imported macros is then generated.

After that the library's code is generated in its original form. No further processing is needed by the `define-library` macro because the compiler will use the namespace declarations to map the names appropriately during the compilation process.

4.3 Other features

In this section we explain other features of the module system that are not essential for reaching our goal but that are useful in day-to-day use.

4.3.1 Optional version. During the development phase it is good to have a fast turnaround time when debugging a library. It would be tedious to assign a new version after each change of the code. For this reason the module system distinguishes *installed* libraries from those that are *not installed*. When a local copy of a library has been obtained from a version controlled repository (on the network or the local filesystem) it is *installed* and the different versions can be referenced in `import` declarations. A library that is stored in a local directory, possibly managed by a version control system, is *not installed*. In this case `import` declarations must not refer to a specific version and the current state of the code is used. This improves the workflow as it avoids having to install the library after each change. Nevertheless, if it is in a version controlled repository, it is possible to install it whenever there is a need to assign a version to it.

4.3.2 Module aliases. With the `(define-module-alias` *`lib1 lib2`*`)` form, symbolic names can be defined to refer to libraries and library references can be redirected to other locations. This is useful for the development phase for quickly swapping one library for another. It also allows putting the choice of library versions in a centralized place instead of each and every `import` declaration to be able to upgrade to new versions with a single edit. For example, when the following definitions are in effect:

```
(define-module-alias (gitlab.com/zoo cats)
                     (gitlab.com/zoo cats @2.0))

(define-module-alias (fh)
                     (github.com/fred hello))
```

an `(import (gitlab.com/zoo cats))` will import the library `(gitlab.com/zoo cats @2.0)` and an `(import (fh @1.0))` will import the library `(github.com/fred hello @1.0)`. The module alias definitions that are put at the root of a library's directory in the file `_setup_.scm` of a directory will apply automatically to the libraries in that directory (with lexical

scoping rules respecting the nesting of the directories up to the root of the repository).

4.3.3 Library management. Automatically installing a library from the network when it is referenced in a deserialization could be a security issue. The Gambit interpreter allows manual installation of modules through command-line arguments, for example `gsi -install github.com/fred/hello`. Moreover, the runtime system maintains a whitelist of the locations from which libraries are unconditionally installed. By default the whitelist contains only `github.com/gambit`, the Gambit project account. The whitelist can be extended through environment variables and command line arguments. When the library's location is not on the whitelist a confirmation will be asked of the user if a REPL is currently started (the runtime system can be configured to always ask for confirmation, or to always refuse to install such libraries).

5 EVALUATION

The original implementation of Termite Scheme was able to send messages containing code that the receiving node did not have as long as the code was interpreted. Our module system extends this capability to compiled code. In this section we evaluate experimentally the performance gain due to the more compact messages and the faster compiled code.

Three machines with different operating systems and architectures were used in the experiments to exercice the machine independence. All three machines are on the same Gigabit ethernet LAN. The machine $M_{ARM/Linux}$ is a 4-core armv7l with 2GB RAM running Linux 4.19 (Raspberry Pi). The machine $M_{x86/macOS}$ is a 6-core Intel i7-8700B with 32GB RAM running macOS. The machine $M_{x86/Linux}$ is a water-cooled 4-core Intel i7-7700K with 16GB RAM running Linux 4.9. The later machine is the fastest and the execution time measurements are the most stable of the three machines due to the better thermal control. This fast machine is always used as the destination of the code migrations; a situation that is representative of the case where the destination of a RPC is a fast compute server.

Three standard Scheme benchmark programs of different source code sizes (in bytes) were used to see the effect of the code size:

- Puzzle (4K)
- Scheme (40K)
- Compiler (400K)

The internal iteration count of the programs was adjusted so that they would have an execution time when interpreted that is roughly proportional to their size. So 400K has both the largest code size and the longest run time (roughly 10 seconds on $M_{x86/Linux}$).

The programs are adapted to our use case as follows. The program is turned into a library by wrapping it in a `define-library` form that exports the program's main entry point and the library is put in a repository hosted on `github.com`. A separate driver program simply imports the library and then causes the program to be executed

$M_{x86/macOS}$

Time (ms)	4K		40K		400K	
Total for RPC	146.4	± 0.6	966.9	± 6.1	10463.8	± 3.3
On destination	131.6	± 0.2	948.7	± 5.9	10381.0	± 2.7

$M_{ARM/Linux}$

Time (ms)	4K		40K		400K	
Total for RPC	179.2	± 1.6	1002.7	± 8.1	10801.1	± 11.0
On destination	132.6	± 0.7	954.6	± 0.0	10390.5	± 2.6

Figure 7: Timings in the INTERPRETED scenario with $M_{x86/macOS}$ or $M_{ARM/Linux}$ as the start node

$M_{x86/macOS}$

Time (ms)	4K		40K		400K	
Total for RPC	15.5	± 0.7	48.5	± 0.6	478.7	± 0.6
On destination	2.2	± 0.0	35.2	± 0.1	463.3	± 0.3

$M_{ARM/Linux}$

Time (ms)	4K		40K		400K	
Total for RPC	26.9	± 1.2	60.4	± 0.6	492.5	± 0.7
On destination	2.2	± 0.0	35.2	± 0.0	462.8	± 0.3

Figure 8: Timings in the STEADY-STATE scenario with $M_{x86/macOS}$ or $M_{ARM/Linux}$ as the start node

$M_{x86/macOS}$

Time (ms)	4K	40K	400K
Total for RPC	1208.6	2460.3	148536.2
On destination	2.2	36.1	464.7

$M_{ARM/Linux}$

Time (ms)	4K	40K	400K
Total for RPC	1159.8	2502.0	153272.6
On destination	2.2	37.1	464.1

Figure 9: Timings in the FIRST-INSTALL scenario with $M_{x86/macOS}$ or $M_{ARM/Linux}$ as the start node

on $M_{x86/Linux}$ using Termite's `on` form calling the library's "main". The total execution time is measured and also the execution time on the destination node. The difference between these measures is accounted for by the message transfers, the serialization and deserialization, and when the library isn't currently installed, the installation of the library source code locally from `github.com`, the Scheme to C compilation, and the C compilation. The programs are run 20 times, the top and bottom outliers are removed to account for random variations in the network latency, and the tables of results contain the average and standard deviation for each measure in milliseconds.

Three scenarios are tested:

- **INTERPRETED**: The whole program is interpreted. This represents the performance achievable before our module system was implemented.
- **FIRST-INSTALL**: The compiled program is executed and the destination machine is installing the library for the first time.

- **STEADY-STATE**: The compiled program is executed and the destination machine has previously installed and compiled the library.

Figures 7, 8, and 9 give the timings for the INTERPRETED, STEADY-STATE, and FIRST-INSTALL scenarios respectively with either $M_{x86/macOS}$ or $M_{ARM/Linux}$ as the start node, and always $M_{x86/Linux}$ as the destination node.

The first observation is that in all the scenarios the speed of the start node has very little bearing on the total RPC time. This is to be expected because most of the work is done on the destination node. The slower times are generally for $M_{ARM/Linux}$. This can be explained by the higher messaging overhead.

The message transfer overhead (network latency and serialization/deserialization) will increase with the size of the messages. In the STEADY-STATE scenario the messages are the parameters of the called procedure and the result. This goes from less than a hundred bytes (for 4K) to tens of kilobytes (for 400K). In this scenario the messaging overhead varies between 13-15ms for $M_{x86/macOS}$ and between 25-30ms for $M_{ARM/Linux}$ (slower processor and network interface). The execution time on the destination is essentially identical. In the INTERPRETED scenario the messages also carry a representation of the code to be executed, which is large for 400K. In this scenario the messaging overhead varies between 15-83ms for $M_{x86/macOS}$ and between 47-411ms for $M_{ARM/Linux}$ (here again the overhead is impacted by the slower processor and network interface). Nevertheless the messaging overhead represents a small fraction of the total RPC time. For the STEADY-STATE scenario, as expected the messaging overhead is at its highest for 4K, the shortest running program, because messaging represents more work than the actual computation on the destination.

To evaluate the speed improvement of RPC calls achieved with our module system, we can compare the INTERPRETED and STEADY-STATE scenarios. The total RPC time for STEADY-STATE is up to 22x faster for $M_{x86/macOS}$ and $M_{ARM/Linux}$ on 400K. The shortest running program, 4K, has the lowest but still considerable speedup of 6x-9x.

Figures 9 shows that the time for the installation and compilation of the library can be quite large for large programs (400K, which is about 10,000 LOC, takes about 150 seconds and the others less than 2.5 seconds). Thankfully, installation only has to be done once per library and version used and libraries are rarely so big, especially when good modular programming practices are used.

6 RELATED WORK

The package management offered by our system supports installing multiple versions of a package, which ensures the same dependencies on all nodes. The package management of the Go [6] programming language supports installing and using multiple versions of a module, but it does not support execution time installation of modules that is needed for hot code update. QuickLisp [27] follows a different approach of only keeping the last installed version of packages, which can break dependencies. The module identification does not include the location as the module names are mapped to their location using a central directory, which means the module names have to be registered to avoid name clashes. Like our system, QuickLisp stores modules on public VCS services and has an automatic installation of modules but it is not tied to deserialization. Nix [12] is a system package manager that shares the same idea of keeping multiple versions of a package to avoid breaking dependencies. However, it is not meant to be used as a language package manager and thus it is not integrated with a specific language. Erlang supports hot code update but because the module identification does not contain the location of the library the installation of modules must be done separately.

Before the R6RS standard was ratified (2007), most Scheme systems had designed their custom module system. Support for R6RS and its module system was added to some systems notably Chez Scheme, Guile, Larceny, and PLT Scheme (now Racket). The R6RS standard includes a `library` form to define libraries which has much syntactic similarity to the R7RS `define-library` form used by our work. A relevant difference is R6RS' support for version information in the library name. Our module system allows version information in `import` declarations but not in the `define-library` form. We believe it is less error prone to obtain this information from the underlying version control system as it avoids possible inconsistencies with the code.

Since the ratification of R7RS (2017), support for its less complex module system is growing among Scheme systems with close to 20 systems supporting it. For the strictly R6RS compliant ones the Akku.scm project [26] has developed a converter from the R7RS `define-library` form to the R6RS `library` form. The more widespread support for R7RS' module system is one of the motivations for adopting it in our work.

Over the years different groups have implemented module systems extending Gambit Scheme. Black Hole [1] is an R5RS compatible module system designed to add as little extra syntax as possible to Gambit Scheme. JazzScheme [8] and Gerbil [25] are more intrusive as they promote a whole new custom program structure and syntax, and add features such as object orientation. The SchemeSpheres [2] project has used a prototype implementation of our `define-library` macro, so it has much similarity to our latest work.

However, none of these systems offer the same combination of features as our work and specifically none offers transparent deserialization of compiled procedures and continuations.

The library naming approach we have used which includes the repository location to identify the library could be used by other R7RS Scheme systems without modifying their implementation. This would help avoid library name clashes and also pave the way for future extensions of those Scheme systems to automatically install the library from a public repository (independent of any support for procedure and continuation deserialization).

ACKNOWLEDGMENTS

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

REFERENCES

[1] Black Hole, a R5RS compatible module system for gambit. https://github.com/per-gron/blackhole, 2019. Accessed: 2020-02-21.

[2] Schemespheres. https://github.com/alvatar/spheres, 2019. Accessed: 2020-02-21.

[3] David Alan and David Alan Halls. Applying mobile code to distributed systems. Technical report, 1997.

[4] Joe Armstrong. A history of erlang. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, page 6–1–6–26, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595937667. doi: 10.1145/1238844.1238850. URL https://doi.org/10.1145/1238844.1238850.

[5] Andrew Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, 1984. doi: 10.1145/2080.357392. URL http://doi.acm.org/10.1145/2080.357392.

[6] Tyler Bui-Palsulich and Eno Compton. Go reference manual 1.13.5. https://blog.golang.org/using-go-modules, 2019.

[7] Luca Cardelli. The functional abstract machine, 1983.

[8] Guillaume Cartier and Louis-Julien Guillemette. Jazzscheme: Evolution of a lisp-based development system. In *2010 Workshop on Scheme and Functional Programming*, page 50. Citeseer, 2010.

[9] Henry Cejtin, Suresh Jagannathan, and Richard Kelsey. Higher-order distributed objects. *ACM Trans. Program. Lang. Syst.*, 17(5):704–739, September 1995. ISSN 0164-0925. doi: 10.1145/213978.213986. URL https://doi.org/10.1145/213978.213986.

[10] William D. Clinger, Anne H. Hartheimer, and Eric M. Ost. Implementation strategies for first-class continuations. *Higher-Order and Symbolic Computation*, 12(1):7–45, Apr 1999. ISSN 1573-0557. doi: 10.1023/A:1010016816429. URL https://doi.org/10.1023/A:1010016816429.

[11] Pavel Curtis and James Rauen. A module system for scheme. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 13–19, New York, NY, USA, 1990. ACM. ISBN 0-89791-368-X. doi: 10.1145/91556.91573. URL http://doi.acm.org/10.1145/91556.91573.

[12] Eelco Dolstra, Merijn Jonge, and Eelco Visser. Nix: A Safe and Policy-Free System for Software Deployment. pages 79–92, 01 2004.

[13] Marc Feeley. Compiling for multi-language task migration. *SIGPLAN Not.*, 51(2):63–77, October 2015. ISSN 0362-1340. doi: 10.1145/2936313.2816713. URL http://doi.acm.org/10.1145/2936313.2816713.

[14] Marc Feeley. Gambit v4.9.3. Reference Manual, 2 2019. URL http://www.iro.umontreal.ca/~gambit/doc/gambit.pdf.

[15] Marc Feeley and Philip W. Trinder, editors. *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang, Portland, Oregon, USA, September 16, 2006*, 2006. ACM. ISBN 1-59593-490-1.

[16] Adrian Francalanza and Tyron Zerafa. Code management automation for Erlang remote actors. In Jamali et al. [21], pages 13–18. ISBN 978-1-4503-2602-5. doi: 10.1145/2541329.2541344. URL https://doi.org/10.1145/2541329.2541344.

[17] Matthew Daniel Fuchs. *Dreme: For Life in the Net.* PhD thesis, USA, 1995. AAI9609199.

[18] Stefan Fünfrocken. Transparent migration of java-based mobile agents: Capturing and re-establishing the state of java programs. *Personal Technologies*, 2(2):109–116, Jun 1998. ISSN 1617-4917. doi: 10.1007/BF01324941. URL https://doi.org/10.1007/BF01324941.

[19] Guillaume Germain. Concurrency oriented programming in Termite Scheme. In Feeley and Trinder [15], page 20. ISBN 1-59593-490-1. doi: 10.1145/1159789.1159795. URL http://doi.acm.org/10.1145/1159789.1159795.

[20] Robert S. Gray. Agent tcl: A flexible and secure mobile-agent system. In *Proceedings of the 4th Conference on USENIX Tcl/Tk Workshop, 1996 - Volume 4*, TCLTK'96, pages 9–23, Berkeley, CA, USA, 1996. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267498.1267500.

[21] Nadeem Jamali, Alessandro Ricci, Gera Weiss, and Akinori Yonezawa, editors. *Proceedings of the 2013 Workshop on Programming based on Actors, Agents, and Decentralized Control, AGERE!@SPLASH 2013, Indianapolis, IN, USA, October 27-28, 2013*, 2013. ACM. ISBN 978-1-4503-2602-5. URL http://dl.acm.org/citation.cfm?id=2541329.

[22] A. Lukić, N. Luburić, M. Vidaković, and M. Holbl. Development of multi-agent framework in JavaScript. In *ICIST 2017 Proceedings Vol.1*, pages 261–265, 2017.

[23] Stefan M, Raymond Bimazubute, and Herbert Stoyan. Mobile intelligent Agents in Erlang. In *Fourth International ICSC Symposium on ENGINEERING OF INTELLIGENT SYSTEMS (EIS 2004)*, 2004.

[24] Alex Shinn, John Cowan, and Arthur A. Gleckler. Revised^7 report on the algorithmic language Scheme. 2017.

[25] Dimitris Vyzovitis. Gerbil scheme. https://cons.io/, 2020. Accessed: 2020-02-21.

[26] Göran Weinholt. Akku package management made easy. https://akkuscm.org/, 2019. Accessed: 2020-02-21.

[27] Zach Beane. QuickLisp. https://github.com/quicklisp, 2020. Accessed: 2020-04-02.