This paper will appear in the preceedings of the Usenix Tcl/Tk Workshop '95.

An Anatomy of Guile
The Interface to Tcl/Tk

Thomas Lord
lord@cygnus.com
Cygnus Support

Abstract

Guile is an extension language library *consisting of a virtual machine, run-time system, and front ends for multiple languages. Guile has been closely integrated with Tcl/Tk* [**Ousterhout**] *so that Tcl and Tk modules can be used by Guile programs, and Guile programs can be used to extend Tcl/Tk applications.*

This paper gives an overview of the structure and function of Guile, and includes some notes about how it is expected to evolve in the future. The technical relation between Tcl/Tk and Guile is given special attention.

What is Guile?

Guile is the GNU project's *extension language library*.

`Libguile` provides support for multiple, integrated extension languages sharing a common object system, calling conventions, and libraries of extension code. The library is organized around a flexible implementation of Scheme. It is designed to mix unobtrusively with ordinary C programs.

Why build an extension language library that supports more than a single extension language? There are several reasons:

...so that users will have a choice of language.

A user might prefer one language over another, or one language might be more appropriate to a situation than another. For example, even though `libguile` is being derived from a Scheme interpreter, one goal of the project is to make Guile suitable for use in GNU Emacs. There are many Emacs Lisp programmers and programs, and a considerable amount of documentation – so in addition to Scheme, full support for

Emacs Lisp is very important. Another example: some users dislike Lisp dialects, and so a more C-like language is a good idea.

...so that extension writers can combine multiple bodies of code.

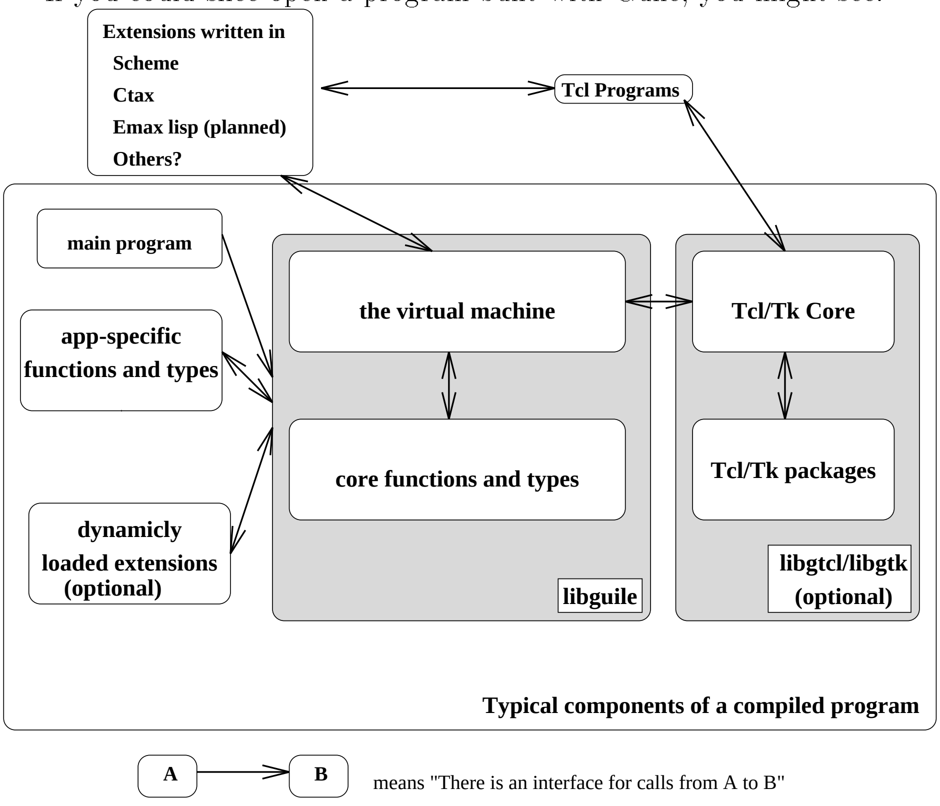
For example, the GNU project's World Wide Web browser, TkWWW is currently written in a mixture of Scheme and Tcl. TkWWW was originally written in pure Tcl, and is now being extending drawing on code from the popular Scheme library **slib**. Eventually, TkWWW will be able to run programs from the Emacs library as well.

...because it is better to optimize and port one multi-lingual interpreter than several monolingual interpreters.

Under the hood, the implementations of many languages are quite similar. There is a lot of duplicated effort in implementing them separately.

An Anatomy of Guile

If you could slice open a program built with Guile, you might see:



Schematic overview of a Guile-based application

A single virtual machine and run-time supports a variety of languages. The virtual machine is inter-operable with Tcl/Tk and consequently, so are Tcl programs and programs interpreted by the virtual machine.

A particular application can define new additions to the built-in run-time. Other additions can be dynamicly loaded – including additions which are extension language programs compiled to native code.

In the next several sections, major features of Guile will be individually described.

Core Functions and Types

`libguile` includes a core set of built-in types and procedures that are accessible from extension languages. Included are the core procedures of standard Scheme, a Posix system call interface, a regular expression package, an object system, and more.

This section will describe (omitting many details) the representation and interface to objects understood by the Guile virtual machine, and the C calling conventions that apply to built-in functions. Most of the details of representation mentioned in this section are helpful for understanding Guile, but are hidden by the public interfaces to `libguile`. This is a “behind the curtain” view.

- **Guile Objects**

All Guile objects are declared in C to be of the type `SCM`. This type is opaque and values of type `SCM` are manipulated almost entirely by functions and macros. There are two important exceptions: these objects can be compared for equality (`eq?` in Scheme) using C's `==` and `!=`; these objects can be assigned to using C's `=`. The size of an `SCM` object is guaranteed to be the same as a `void *` object.

Internally to `libguile`, `SCM` values fall broadly into two classes: immediates and non-immediates. An immediate object fits entirely within an `SCM` variable and represents an immutable value. Some examples of immediate values are character constants and small integers. A non-immediate object is one that is represented by a pointer into the *cons pair heap*. The pointer points to a *cons pair*. The cons pair heap is itself one or more large regions of memory allocated by `malloc`, carved up into individual pairs.

A cons pair is heap memory containing two `SCM` values. In keeping with tradition, these values are called the CAR and CDR of the pair. A pair may be used in any of several ways. It may be part of a traditional lisp list in which case the CAR is the first element of the list and the CDR is the remaining sub-list. Or, a pair may hold some other composite type, usually storing type and length information in the CAR and type-specific data in the CDR.

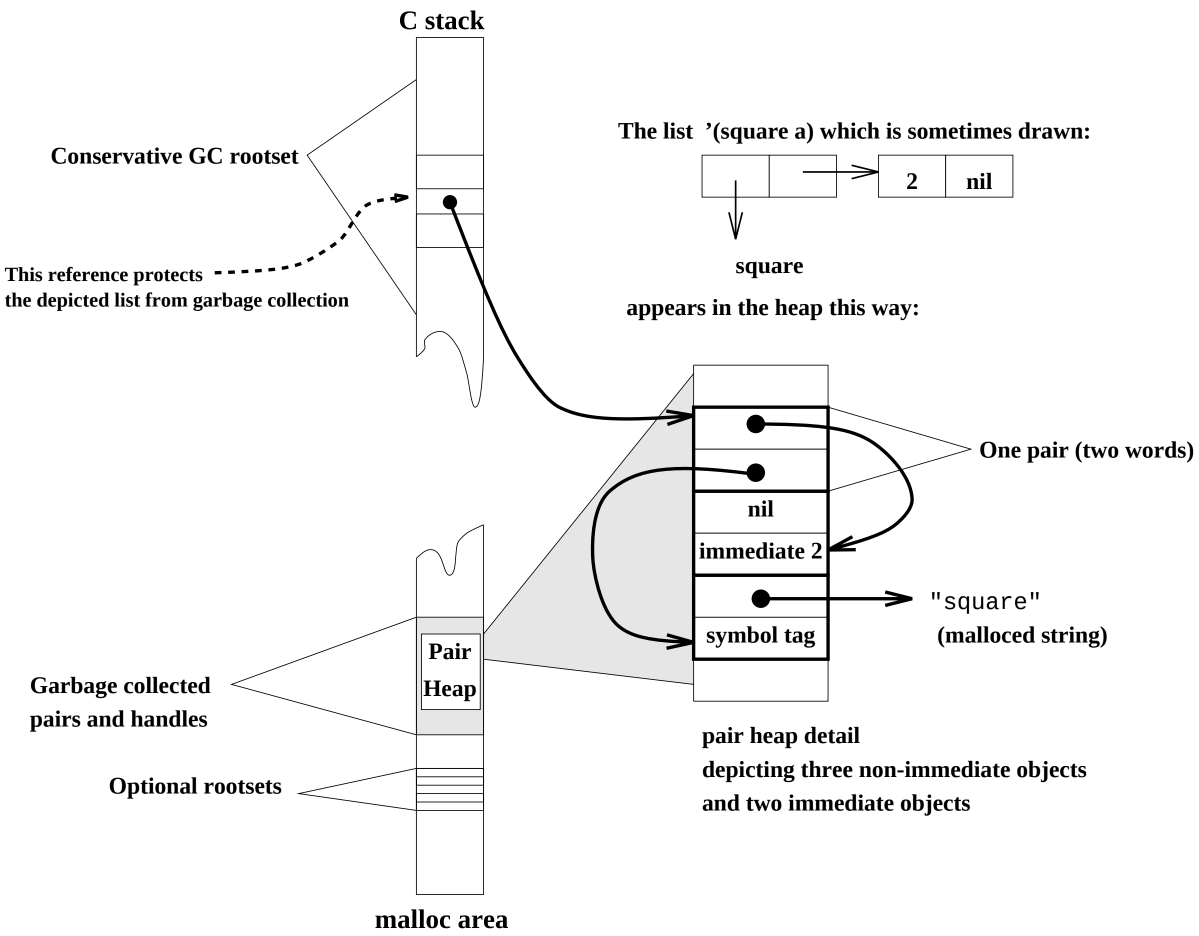
All values, immediate and non-immediate, are marked with their type. Typically, type information is stored in the low order bits of immediate values and the CAR of non-immediate values.

CAR	CDR	
first elemt	remaining sublist	a list
length	malloced (char *)	a string or symbol
length	malloced (SCM *)	a generic array
malloced (void *)	malloced (void *)	struct type (car points to a vtable, cdr to data)
type information	(float)	floating point number
type info	SCM (*)()	built-in-procedure
type info	(SCM) or (void *)	interpreted code
code (SCM)	environment (SCM)	Scheme closure

Some Formats of Pairs (tag bits not shown)

Built-in non-immediate types include lists and arrays (Scheme `pairs` and `vectors`), abstract streams (Scheme `ports`), arbitrary precision integers, complex numbers, regular expressions, character strings and more.

Non-immediate objects are automatically garbage collected. There is very little that programmers ever need to do to make the garbage collector work correctly. The garbage collector automatically searches the C stack (and also searches any other regions of memory explicitly designated) for apparent pointers into the pair-cell heap. All non-immediates that appear to be referenced in that way are safe from garbage collection. Any object that is referenced directly or indirectly by a saved object is itself saved. Any remaining objects are reclaimed and recycled because no legitimate manipulation of values known to the program will again reference those objects. Here is a typical memory map of a Guile-based program:



The Cons Pair Heap

Programmers using `libguile` can define new types of non-immediate objects. This is explained further in a later section.

- **Built-in Procedures**

Built-in procedures are available both to C programmers using `libguile` and to programmers calling these procedures from an extension language. The built-in procedures are the basis set of operations that can be performed on Guile objects.

`libguile` includes a superset of the procedures required by standard Scheme [**Clinger**]. Additional optional packages include an advanced array package (volunteers are working on implementing a library of APL-like array functions), an interface to Posix system calls and an interface to the GNU regular expression library.

For the most part, C functions implementing Guile procedures follow normal C calling conventions. For example, the procedure `cons`, that takes two values and constructs a new pair from them, can be declared this way:

```
extern SCM gscm_cons (SCM car, SCM cdr);
```

Procedures that take a variable number of arguments generally expect those arguments to be passed as a list (a linked list of cons pairs). For example, the `apply` procedure applies an object that represents a procedure to an arbitrary number of arguments. To apply a procedure object to three arguments, one might write:

```
SCM result = gscm_apply (proc_object,
                         gscm_listify (a, b, c, GSCM_EOL_MARKER));
```

`gscm_listify` is the sole `libguile` function that takes a variable number of arguments in the usual C fashion.

If an error occurs in a built-in function, a `libguile` exception is thrown. Usually this results in a `longjmp` past the caller to an error handler, although an interface exists for preventing this by posting an overriding exception handler.

The Virtual Machine

In order to provide support for an arbitrary number of extension languages, `libguile` implements a virtual machine that interprets a shared language to which extension languages can be translated ([**Sah**]).

- **The Graph Interpreter**

The primary Guile virtual machine interprets code stored in graphs of cons pairs. As it interprets, it rewrites parts of the graph for efficiency. For example, upon the first evaluation, a symbolic variable reference is rewritten to a fast virtual instruction that uses the resolved location of the named variable. In this way, a variable lookup is eliminated from subsequent executions of the same part of the code graph.

The virtual machine implements support for *memoizing macros*. A memoizing macro is a special kind of procedure that rewrites one expression to form another. When it occurs in code, the macro is evaluated, and then its return value is itself evaluated.

A memoizing macro is special because when a call to one is first evaluated, the result is saved and that particular call is never re-evaluated. The return value of a memoizing macro will be evaluated every time the interpreter passes through that region of the code graph – but the macro itself, that generated that return value, will only be evaluated once. Essentially, memoizing macros are a mechanism for incremental, demand-driven code generation.

Interpreted Scheme is implemented for this virtual machine mostly by a series of memoizing macros. The Scheme macros transform cons pair syntax-trees of Scheme source into cons pair trees of internal virtual machine instructions. The transformation is incremental and demand driven: loading new source code into the system is very fast in exchange for slower execution the first time through any particular code-path.

Other languages can be implemented for the `libguile` interpreter by translating them to Scheme and interpreting that. The memoizing macro facility makes it possible to do much of the translation lazily – perhaps from a cons pair syntax tree of the source language program.

If a language **L** can be successfully translated to Scheme, the implementor can avoid the harder task of writing a full implementation of **L**. By rendezvousing at (an extended) Scheme, various implementors can make their languages inter-operable. Once some **L** can be translated to Scheme, it can be compiled to native code via a **Scheme->C** translator (such as the Guile-compatible translator `hobbit` ([**Tammet**])) and a **C** compiler. ([**Steele**])

- **The Future Byte-Code Interpreter**

An extension to the Guile VM, a *byte-code interpreter* (something of a misnomer in this case), is under construction and is *likely* to be in a useful condition by the end of 1995. (Then again, GNU is a non-prophet project [**RMS**], so who can say for sure.) The byte-code interpreter and the graph interpreter will interact smoothly – evaluation can transparently be of a mix of byte-code and graphs. An experimental version of the byte-code interpreter is included with snapshots of Guile.

The byte-code interpreter executes one-word virtual instructions stored along with operands in a contiguous, typically malloced, array. It uses an instruction-set derived from, but not strictly compatible with, the specification of the Java VM [**Sun**].

The byte-code interpreter is being built in order to provide:

a portable, fast-loading, compact representation for programs

a way to trade off time spent compiling extension programs for faster execution

an alternative target to Scheme for language implementors

a target for languages that use native-format (untagged) numbers and structures

Other systems have already demonstrated the value of a byte-code interpreter for its convenient code representation and good performance ([**Emacs**]).

Because the byte-code and graph interpreters will work together, language implementors will have a choice of targets. For some languages (for example, `ctax` which is described below), the byte-code assembly language may be a more reasonable target language.

The byte-code instruction-set includes instructions that operate on intermediate values and structure fields stored not as `SCM` objects, but in native format (for example: untagged, normal, integers). This is convenient for implementing, with excellent performance, extension languages which are more like C or C++.

The Interface to Application Specific Functions and Types

Programmers using `libguile` can define new built-in procedures and types for their application. Built-in procedures are defined as ordinary C functions, taking arguments and returning a result of type `SCM`. These functions are declared to the interpreter at any time (usually during program initialization).

New types may also be declared to the interpreter. Programmers must specify a name and size for the type. Optionally they may specify a print function for the type, an equivalence test (for Scheme `equal?`), and a clean up function that is called when an object of the new type is freed by the garbage collector. Generally, new types are useless in and of themselves but are made useful by also defining new procedures to construct and manipulate them.

Application-specific types are all non-immediate values. They are represented by a cons pair handle and a malloced block of data. The format of the malloced data is arbitrary and may contain fields of type SCM. The malloced data will be searched by the garbage collector – programmer's do not need to write a "GC stub function" for a new type.

the procedure name

documentation

```
/* A named procedure. */
procedure_obj = gscm_define_procedure ("draw-line", draw_line_fn, 5, 0, 1, draw_line_doc);
```

five required arguments
zero optional arguments
yes, a var-args procedure

```
/* An anonymous procedure. */
procedure_obj = gscm_make_subr (draw_line_fn, 5, 0, 1, draw_line_doc);
```

implicit first argument passed by the new procedure

```
/* Creating a closure by currying. */
new_procedure_obj = gscm_curry (procedure_obj, some_val);
```

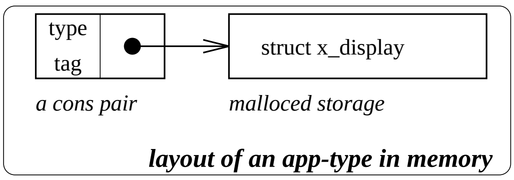
existing procedure of at least one argument

Constructing named and anonymous procedures

```
/* Structure describing an application specific type: */
struct gscm_type x_display_type =
{
  0,
  "X-display",
  print_x_display,
  free_x_display
};
```

This structure is referenced by libguile when managing objects of the new type (called "X-display").

This structure defines the layout of the malloced part of an X-display object.




```
/* How display objects might be represented */
struct x_display =
{
 DISPLAY d; /* the X-related state */
  SCM properties;  /* a property list for extension programs */
};
/* Allocating an app-specific type: */
SCM obj = gscm_alloc (&x_display_type, sizeof (struct x_display));
/* Accessing the state of an app-specific type: */
struct x_display * x_dpy = (struct x_display *)gscm_unwrap (&x_display_type, &obj);
```

Application-defined types

Guile Extension Languages

The Guile virtual machine is a platform for multiple extension languages. This section will discuss what languages are available and planned and briefly how they are implemented.

- **Modified and Standard Scheme**

Historicly, the bulk of the code in `libguile` was written as a Scheme interpreter ([**Jaffer**]). Scheme, because of its simplicity and generality remains the language around which Guile is organized. Scheme serves as a systems programming language for the Guile virtual machine.

A large body of Scheme code exists that Guile is able to run. A notable example is the popular Scheme library **slib** ([**Jaffer2**]). Another example is **hobbit**, a Guile-compatible Scheme->C compiler, capable of producing dynamicly loadable object files ([**Tammet**]).

The Guile version of Scheme differs from standard Scheme ([**Clinger**]) in two ways. First, in Guile Scheme, symbols are case sensitive. Second, in Guile Scheme, there is no distinction made between the empty list and boolean false (between `'()` and `#f`).

Although large bodies of real-world Scheme code work without modification in the Guile dialect, some picky Scheme code will not. Therefore, support for running programs in strictly standard Scheme is planned for a future version of Guile.

As explained earlier, Scheme is implemented on top of the virtual machine by means of memoizing macros that incrementally translate Scheme source code (stored as lists) into internal code (also stored as lists).

It is handy that the interpreter accepts source code as lists because it means that extension language programs can generate and evaluate Scheme expressions on-the-fly. An example of such a program is the `stand-alone-repl` (*Read Eval Print Loop*) that interactively reads and evaluates Scheme expressions.

- **Ctax**

Guile also comes with an implementation of a language called ctax. Ctax is Scheme, but with a C-like syntax.

Here is an example of a ctax program and a Scheme program to which it can be compared:

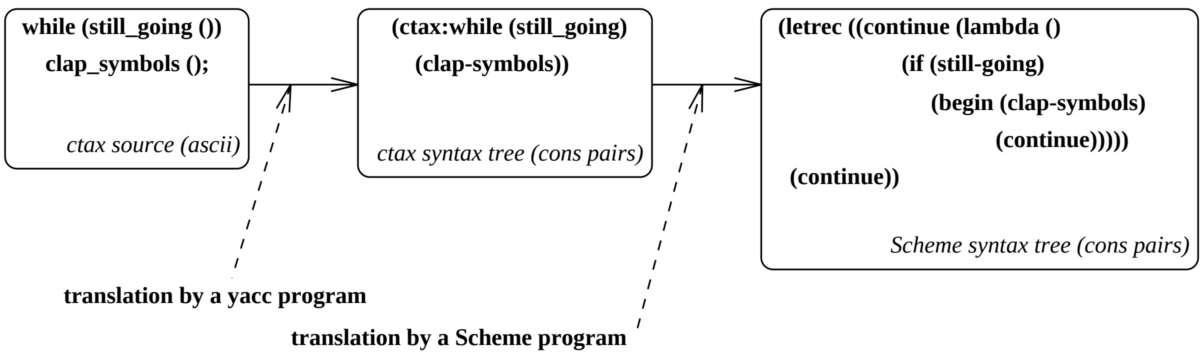
```
/* Multiply all numbers between m and n */
scm
product (m, n)
{
  scm answer;
  scm x;

  answer = 1;
  for (x = m; x <= n; ++x)
    answer *= x;
  return answer;
}

;; And again in Scheme:
;;
(define (product m n)
  (let loop ((answer 1)
             (x m))
     (if (<= x n)
         (loop (* answer x) (+ x 1))
         answer)))
```

Comparing `ctax` and `Scheme`

As this paper is being written, ctax is implemented by means of a translation to Scheme. A simple parser produces syntax trees as lists, these are translated to Scheme and evaluated:



Stages in the translation of Ctax

In the future, ctax will be extended so that variables can be declared to be of specific types (in addition to the universal type `scm`). A `ctax->byte-code` translator is in the works that will take advantage of the byte-code interpreter's native-format math instructions. Support for object oriented programming is in the works as well.

- **Emacs Lisp**

Future releases of Guile will support another dialect of lisp: Emacs Lisp. This will allow `libguile` to be used in the implementation of GNU Emacs and it will allow any program linked with Guile to execute Emacs Lisp programs ([**Krawitz**]).

Most of the low-level support for Emacs Lisp has already been built. The remaining hard part is to add an efficient yet flexible implementation of shallow binding – Emacs Lisp is an idiosyncratically dynamicly scoped language. Once a reasonable dynamic binding mechanism has been implemented, translation of Emacs Lisp to Guile Scheme will be a straightforward syntactic transformation.

Combining Languages Via Modules

Extension language programs communicate with the run-time system and with other extension language programs via shared global variables. For example, an extension package that defines a new data structure might publish global definitions for the constructors and accessors for that data structure.

Guile includes a *user level module system* that provides a fairly conventional access-control interface for managing global bindings. User level modules specify public and private definitions. Programs can specify which modules' public namespaces are to be searched for globals. An integrated autoloader loads definitions of requested modules on-demand.

The user level module system is built on a hook called the *low level module system*. Low level modules are a simple but powerful mechanism that, as will be explained, play an important role in the interface between Guile and Tcl. Low level modules work as follows:

Guile defines a single, special, global namespace called the `symhash` table. Some definitions in this namespace have special meaning to the interpreter. For example, extension-language signal handlers can be specified by defining particular names in the symhash namespace.

One special name in the symhash namespace is the variable `*top-level-lookup-thunk*` which may optionally be bound to a procedure. If it is bound, then that procedure is used to map variable names to global variables.

Storage cells for global variables are first-class anonymous objects that may be explicitly created and manipulated. When the interpreter is interpreting a program, it may find an unresolved variable name that needs to be looked up in the appropriate namespace. If that happens, then the procedure that was the value of `*top-level-lookup-thunk*` at the time the code being evaluated was loaded is called, passing the unresolved variable name as an argument. The lookup-thunk may return storage for a global, thereby resolving that name, or it may return false to signal an error. In short, the 'linker' that resolves global symbolic references in extension programs is entirely customizable.

To illustrate the power of the `*top-level-lookup-thunk*`, here is some code that implements a simple kind of 'safe execution' environment. The goal of this code is to transform the interpreter into one in which only the definitions in an approved list are available. The list of approved definitions would presumably omit dangerous functions such as most system calls. If Scheme code leaves you cold, don't sweat it – the details of this example are not central to the paper.

```
;;; The list of approved definitions:
;;;
(define safe-definitions '(list + - * / cons car cdr ... etc))


;;; Establish the rule for how global variables are looked
;;; up:
;;;
(set! *top-level-lookup-thunk*
  (let ((definitions (make-table)))
    (lambda (name defining?)
       (or (aref definitions name)
           (let ((safe? (member name safe-definitions)))
              (and (or safe? defining?)
                   (let ((answer (make-variable
                                   (and safe?
                                        (variable-ref
                                           (built-in-variable name))))))
                     (aset definitions name answer)
                     answer)))))))
```

Safe-Guile **in 12 Lines of Code.**

The Interface to Tcl

Tcl ([**Ousterhout**]) defines at one layer a scripting language in which useful programs have been written, and at another layer, a C call-out mechanism for which useful libraries of C code have been written.

The GNU project wants to adapt a substantial Tcl program, **TkWWW** ([**Wang**]), and Tcl library, **libexpect** ([**Libes**]). So an early priority for the Guile project has been a clean interface to Tcl at both the scripting language and C call-out layers.

Tcl C Call-outs and Guile

The low-level Guile interface to Tcl is based on the public C interface to `libtcl`. Most of the `libtcl` functions have been made available to extension language programs as built-in procedures. Thus, one can write a Scheme program that creates any number of Tcl interpreters, calls `Tcl_Global_Eval` and so forth.

`Tcl_CreateCommand` is among the `libtcl` procedures that can be called from extension programs. For example, one can define a new Tcl command to be not a C function, but a Scheme procedure. By means of this, Tcl programs can call Scheme procedures.

Scheme procedures can call Tcl commands by way of `Tcl_Global_Eval`, but there is a better way. Tcl commands are first-class Scheme objects and can easily be converted to Scheme procedures. When such a procedure is invoked, the corresponding Tcl command is called with no intervening pass through the Tcl string substitution/evaluation mechanism.

```
;;; Creating an interpreter:
;;;
(define interp (tcl-create-interp))

;;; Calling the Tcl evaluator (it returns both status and result):
;;;
(tcl-global-eval interp "expr 5+7")    => (0 . "12")

;;; A tcl command can be converted to a Scheme procedure:
;;;
(define expr (reify-tcl-command interp ´expr))

(expr "5+7")    => 12

;;; Defining a new Tcl command as a Scheme procedure:
;;;
(tcl-create-command interp ´string_smash
   (lambda (a b) (string-append a "<<<SMASH>>>" b)))

(tcl-global-eval interp "string_smash hello world")
    => (0 . "hello<<<SMASH>>>world")
```

Examples of The Low Level Tcl Interface

An Implicit Interpreter

The low-level mechanisms just described provide an awkward way for Scheme to call Tcl and Tcl, Scheme. (Because Scheme is the rendezvous language for Guile, this means that other extension languages can call and be called by Tcl as well.).

Guile also provides a higher-level, specialized but more convenient way for Scheme and Tcl to inter-operate. The high-level mechanism is based on the previously described low-level module system. The high-level Tcl interface uses the module system to transparently import Tcl commands as Scheme procedures. This works much like the earlier 'Safe-Guile' example combined with the low-level Tcl function `reify-tcl-command`, shown in the previous example.

In addition, the high-level interface includes Scheme macros that define a convenient syntax for defining new Tcl commands written in Scheme.

```
;;; Switch-on the high-level tcl interface:
;;;
(set! the-interpreter (tcl-create-interp))
(use-default-tcl-commands)

;;; Now tcl commands are automatically available as procedures
;;; ...no need to call reify-tcl-command.
;;;
;;; Note that set is the Tcl command, unrelated to Scheme set!.
;;;
(set 'a 0)  => 0
(incr 'a)   => 1
(set 'a)    => 1

;;; The high-level syntax for defining new tcl commands
;;; includes a type declaration syntax for arguments.
;;; Declarations are used to automate the conversion of
;;; arguments from strings to other types.  For example:
;;;
(proc average_of_3 ((number a) (number b) (number c))
  (/ (+ a b c) 3))

(tcl-global-eval the-interpreter "average_of_3 1 2 3")
    => (0 . "2")
```

Examples of the High-Level Tcl Interface

The Interface to Tk

The low-level Guile interface to Tk is, like the low level Tcl interface, a subset of the public C functions of `libtk`. At this time, the three functions exported to Guile extension languages are: `tk-init-main-window`, `tk-do-one-event`, and `tk-num-main-windows`.

A small change to `libtk`: canonical commands

A number of Tk-related procedures deal with call-backs: several widget configuration commands, `after`, and `bind`, at least. It has already been shown that Scheme (and consequently other Guile extension languages) can define new Tcl commands – so obviously call-backs can be defined in Scheme by explicitly defining new commands. But there is a better way.

Scheme programmers are accustomed to dealing with *anonymous procedures*. Anonymous procedures are created on the fly and are garbage collected when no-longer needed. They are usually passed around by value, kept in the arguments and local variables of other procedure calls. Sometimes they are stored in other data structures and are garbage collected along with those structures.

Anonymous procedures are ideal for call-backs. They can encapsulate an arbitrary amount of the state that is available at the time of their creation and carry it through to the time the call-back is issued – an effect traditionally achieved in C by storing a call-back as "function pointer plus `void *`".

It is extremely convenient that an anonymous procedure can be garbage collected once it is no longer needed. A named procedure does not have the same benefit – once named, a procedure must persist until the name is explicitly revoked for at any time the system can be asked to produce the procedure given the name. So, asking programmers to name call-backs would impose the burden that programmers would have to remove those names when the call-backs are no longer needed.

The usefulness of anonymous procedures for writing call-backs is considered sufficiently important that some small changes have been made to `libtk` in support of them. These changes add the concept of a *canonical command* to Tk.

A canonical command is a Tcl command related by an automatic naming scheme to some earlier defined command. The command name from which a canonical command's name is derived is called the canonical command's owner. The owner of a canonical command controls it's lifetime – when it is no longer needed by the owner, a canonical command is destroyed.

A canonical command is created by passing a normal command, prefixed by "*", as an argument in a position where a call-back is expected. For example:

```
# An example of canonical commands as seen from wish:
# Start with a normal command:
#
% proc hw {} {puts "hello world"}
% hw
hello world

# Note the asterisk in the command that follows. It means that
# the command name should be canonicalized.  "hw" will be renamed.
#
% pack [.b -text howdy -command *hw]
% hw
invalid command name "hw"

# but if you click the button instead:
hello world

# Which is because the name of the command
# has been canonicalized:
#
% .b configure -command
-command command Command {} {b.__-command }
% b.__-command
hello world

# Since .b owns the canonical command, it
# will go away when .b is done with it.  In this
# case, by setting a new command, the old one is
# made redundant and removed.
#
% .b configure -command {puts "so long, b-dot-underscore-und..."}
% b.__-command
invalid command name "b.__-command"

# And if you click the button...
so long, b-dot-underscore-und...
```

An Illustration of Canonical Commands

The canonical command mechanism allows anonymous procedures to be used as call-backs. When an anonymous procedure is passed as an argument to a Tcl command, a random command name is generated for the procedure. This command name, prefixed by "*", is passed in place of the anonymous procedure. A Tcl command is defined and given the random name. The anonymous procedure is given as the definition of that Tcl command. Because of the "*", the random command is canonicalized. Therefore, the Tcl name for the command is temporary – when the command is no longer needed as a call-back, the name will be removed and the anonymous procedure will be a candidate for garbage collection.

As long as a Tcl command name exists for an otherwise anonymous procedure, it is protected from garbage collection. When the Tcl command name is destroyed (for example a canonical command name that is no longer needed), the anonymous procedure becomes unprotected and if no other references exist, it will be garbage collected.

Gwish

Gwish is a Scheme program modeled loosely on the Scheme interpreter STk ([**Gallesio**]). It builds upon the implicit interpreter interface to Tcl and the low level interface of Tk to present users with a wish-like Scheme shell.

Here some Gwish code for drawing a cheery face. The symbols prefixed by colons are keywords, a Guile extension to standard Scheme. Have a nice day!

```
(canvas ´.c)
(pack ´.c :fill "both" :expand #t)
(.c ´create ´oval  50 5 250 205 :fill ´yellow)
(.c ´create ´oval  110 60 130 110 :fill ´black)
(.c ´create ´oval  170 60 190 110 :fill ´black)
(.c ´create ´polygon 100 125 125 170 150 180 175 170 200 125
                     200 120 175 165 150 175 125 165 100 120
                     :fill ´black :smooth 1)
(bind ´.c ´q (lambda () (destroy ´.)))
```

The TkWWW Project

The GNU project is building a Free ([**Stallman**], [**Stallman2**]) web browser using the Tcl/Tk web browser, TkWWW ([**Wang**]). Volunteers are need to help us quickly bring TkWWW up-to-date with the latest features popular in browsers.

To date, TkWWW has been minimally updated to Tk4 and it runs under Gwish running under on an interpreter linked with the TkWWW ".o" files. Thus, TkWWW is now a Scheme-extensible, ctax-extensible, Tcl-extensible web browser.

The modifications needed to turn TkWWW from a pure Tcl program into a program extensible in Scheme and ctax as well are less than 200 lines of C code, most of which is 'boilerplate' code copied from the sample stand-alone interpreter in the Guile sources. The change took only a few hours to make.

Planned features for TkWWW are in-lined images, an upgrade to a multi-threaded client library, and support for down-loadable, safe extensions.

Conclusions

Guile is a flexible multi-lingual extension language library organized around Scheme and cleanly inter-operable with Tcl/Tk. The performance and generality of Guile can be used to add greater extensibility of any program that already uses Tcl alone. TkWWW is an example of a Tcl program to which Guile has been added.

Guile is organized as a virtual machine and uses Scheme as a "systems programming language" for that machine. A very customizable low-level module system is central to how this VM can be used to run extensions in more than a single extension language.

In particular, a wish-style shell has been written for Guile that provides a convenient interface for writing or extending Tk and Tcl/Tk programs in Scheme or other Guile extension languages.

Acknowledgments and Availability

Its hard to determine just who designed Guile. A large share of the credit surely belongs to Aubrey Jaffer whose excellent Scheme interpreter, SCM, forms the core of the implementation. The module system was designed and built by Miles Bader. Many of the goals of the project were established by Richard Stallman. The byte-code interpreter was inspired by the specification of, and inherits many of the desirable properties of, the Java VM. The architecture was discussed and hacked on by many volunteers. Perhaps it is more a discovery of what we could do with available resources than it is a self-consciously constructed strategy.

For more information about Guile, you can join the Gnu extension language mailing list by sending a subscribe message to **gel-request@cygnus.com**, or you can contact the author directly. Versions of Guile have been in free distribution since February. Generally, snapshots can be found at the ftp location: **ftp.cygnus.com:pub/lord**.

References

[Clinger] William Clinger and Jonathan Rees (Eds.), "Revised^4 Report on the Algorithmic Language Scheme" ftp.cs.indiana.edu:/pub/scheme-repository, 1991

[Emacs] **prep.ai.mit.edu:pub/gnu/emacs-19.28.tar.gz**, the GNU Emacs editor, 1995.

[Gallesio] Erick Gallesio, **kaolin.unice.fr:pub/STk-2.1.6.tar.gz**, the STk Scheme interpreter, 1995

[Jaffer] Aubrey Jaffer, **ftp-swiss.ai.mit.edu:pub/scm/scm4e1.tar.gz**, the SCM Scheme interpreter, 1995.

[Jaffer2] Aubrey Jaffer, **ftp-swiss.ai.mit.edu:pub/scm/slib2a2.tar.gz**, the slib Scheme library, 1995.

[Krawitz] Robert Krawitz et al., "The GNU Emacs Lisp Reference Manual", The Free Software Foundation, 1990

[Libes] Don Libes et al., "libexpect.tar.gz", the expect Tcl library, 1995.

[Ousterhout] John K. Ousterhout, "Tcl and the Tk Toolkit" Addison-Wesley, 1994

[RMS] RMS, personal communication.

[Sah] Adam Sah and Jon Blow, "A New Architecture for the Implementation of Scripting Languages" USENIX Symp. on Very High Level Languages, 1994.

[Stallman] Richard M. Stallman, **ftp.prep.ai.mit.edu:pub/gnu/COPYING**, the GNU Public License, 1991

[Stallman2] Richard M. Stallman, The GNU Manifesto, Free Software Foundation, Cambridge MA, USA, 1993.

[Steele] Guy L. Steele Jr., "Rabbit: A compiler for Scheme" S.M. thesis, Mass. Inst. of Technology, 1978

[Sun] "The Java Virtual Machine Specification", Sun Microsystems, 1995

[Tammet] Tanel Tammet, **ftp.cs.chalmers.se:/pub/users/tammet/hobbit4a.tar.gz**, source code for a Scheme->C compiler, 1995.

[Wang] Joe Wang et al., **tkwww-0.12.tar.gz**, the TkWWW web browser, 1994