Foreign-Function Interfaces for Garbage-Collected Programming Languages

Marcus Crestani

Eberhard-Karls-Universität Tübingen

crestani@informatik.uni-tuebingen.de

Abstract

Programs in high-level, garbage-collected programming languages often need to access libraries that are written in other programming languages. A foreign-function interface provides a high-level language with access to low-level programming languages and negotiates between the inside and the outside world by taking care of the low-level details. In this paper, I provide an overview of what different kinds of foreign-function interfaces are in use in today's implementations of functional programming languages to help decide on the design of a new foreign-function interface for Scheme 48. I have revised several mechanisms and design ideas and compared them on usability, portability, memory consumption and thread safety. I discuss the garbage-collection related parts of foreign-function interfaces using Scheme as the high-level functional language and C as the external language.

1. Introduction

Programs in functional programming languages often need to access libraries that are written in other programming languages. Back in 1996, Scheme 48 [10] received its first foreign-function interface. Over the years, developers connected many external libraries to Scheme 48 using this foreign-function interface. Many other Scheme implementations use a similar foreign-function interface, for example Elk [12], scsh [18], and PLT Scheme's static foreign interface [6]. The foreign-function interface worked reasonably well for a wide variety of external libraries, ranging from database bindings for SQL and ODBC to graphical toolkits like Xlib and Qt, for example.

In 2003, Kelsey and Sperber proposed SRFI 50 [11] to the Scheme community. The proposal describes a foreign-function interface based on the design of Scheme 48's foreign-function interface. After a great deal of discussion that uncovered some weaknesses of the approach, the authors withdrew their proposal [16].

Since then the Scheme 48 developers have tried to connect more external libraries to Scheme 48 and they have realized that the more complex the library the more difficult it was to use the foreign-function interface correctly. In particular, problems with memory management became harder to solve. I then started to look at foreign-function interfaces of other functional programming languages to see if they have come up with solutions to our problems.

Permission to make digital or hard copies of all or part 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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

2008 Workshop on Scheme and Functional Programming

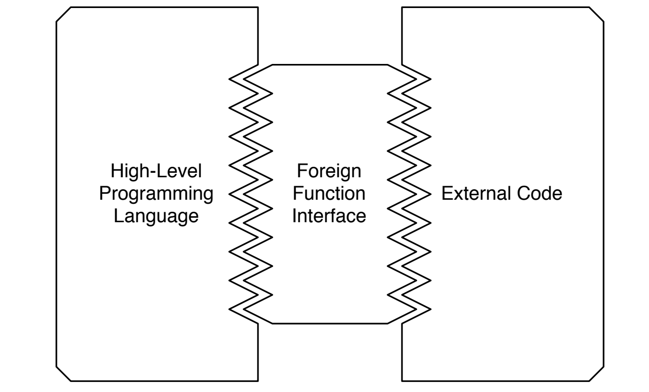

Figure 1. Foreign-function interface

This paper reflects my survey, summarizes the results, and presents Scheme 48's new foreign-function interface, which will replace the current one in the near future.

1.1 Foreign-Function Interfaces

A foreign-function interface provides a high-level programming language with access to other (usually low-level) programming languages and negotiates between the inside and the outside world. Figure 1 illustrates a foreign-function interface that connects the *high-level programming language* with the *external programming language*. A foreign-function interface takes care of all the low-level details that programmers do not need to know when connecting the two worlds:

- converting between different calling conventions such that both worlds can call each other's functions
- marshalling and unmarshalling data and objects between the two worlds
- managing memory and other resources

Most high-level programming language implementations come with a foreign-function interface, usually for C. Naturally, there are many approaches to designing foreign-function interfaces. My goal is to provide an overview of what different kinds of foreign-function interfaces are in use in today's implementations to help decide on the design of a foreign-function interface for Scheme 48. In this paper, I revise several mechanisms and design ideas and compare them on usability, portability, memory consumption and thread safety.

I do not cover tools that automate the generation of glue code for foreign interfaces in this paper. I rather describe the interface on a lower level and show how things are processed there. Once

the low-level details are in place, stub generators can be written to produce code that works with the foreign-function interface.

1.2 Basics

Urban's foreign-function interface survey [23] provides a good overview of many existing foreign-function interfaces, along with discussion of such systems and a list of relevant issues. His survey is large but still far from completion. Therefore, my survey is limited to a few foreign-function interfaces that are in use in functional programming and that seem to be widespread or promising.

Here are the most important issues that foreign-function interfaces have to deal with and that recur in my descriptions of the foreign-function interfaces that this paper covers.

Calling conventions Calling conventions describe the method of calling a function and passing parameters to it, and how to expect the result back from the call. Generally, differences occur on how the caller has to pass the parameters—on the stack, in registers, on the heap, or a mixture—and where the callee puts the result. Another issue are the naming conventions: The high-level and the external language have to negotiate on the naming scheme for each other's functions and variables and how the names are imported into each other's namespaces.

It is desirable to have a foreign-function interface that allows calls in both directions: Calls from the high-level language to the external functions are called *call-outs*. Calls from the external code to high-level functions are called *call-backs*. A common example for call-backs are high-level functions that are used as event handlers for an external GUI library.

There are some complications that arise when mixing call-backs with continuations, especially if a function that is called back from the foreign function captures a continuation that also contains parts of the external function. What should happen if the continuation is invoked after the external function returns and its stack frame is no longer available? Some foreign-function interfaces restrict or forbid the use of continuations in call-backs; Scheme 48, for example, only allows *downward continuations* [10]. Since this paper concentrates on garbage-collection-related issues with foreign-function interfaces, a more detailed answer is outside the scope of the paper.

Another issue is the propagation of errors from the external to the internal world and vice versa. High-level languages use exception mechanisms to propagate errors to the caller or user. External code should be able to use this mechanism as well: It should be able to pass exceptions to the high-level language. The only way to pass exceptions from the high-level language to external code that does not have an exception mechanism is to use parameters and return values.

Marshalling Generally, marshalling means transformation of the memory representation of an object into another data format. Usually, foreign-function interfaces allow the programmer to convert between the data representations of the involved languages. Marshalling may include byte-order changes, boxing and unboxing, or coding-system adjustments, for example. Depending on the interface, the functionality can be available in the high-level language, or in the external code, or both.

Memory management Most high-level languages have a uniform representation for their values, called *descriptors*. Descriptors are objects of high-level languages that are either simple values that fit into a word (i.e. integers) or that are too big to fit into a word and are allocated on the garbage-collected heap. In the latter case, the descriptor represents a pointer into the heap. External code has to coordinate with the garbage collector of the high-level programming language to make sure that all descriptors to which the external code has access are kept live by the garbage collector. If the system uses a moving garbage collector, it has to be able to update the external code's descriptors if it has moved the heap-allocated objects. Therefore, the garbage collector has to know all the locations of descriptors in external code.

Some foreign languages or libraries provide their own automatic memory management: For example, Java or C# come with a garbage collector; the GTK+ graphical toolkit [22] uses a reference-count method to keep track of the objects in use. To correctly interface with an external memory manager, the high-level language must be able to finalize its objects: *Finalization* applies actions to an object when the object is no longer in use by a program. Such an action is called *finalizer*. The finalizer runs after the garbage collector found out that the object is garbage. The foreign-function interface uses the finalizers to communicate to the external memory manager that the object is no longer in use. Although the memory of the external language or library is automatically managed, the foreign-function interface pretends that it is not and manages the memory explicitly. For clarity, I do not explicitly refer to garbage-collected external language in the remainder of this paper.

Thread safety For implementations of high-level programming languages that support multiple system-level threads, it is important that the foreign-function interface guarantees that the system is in a consistent state if a thread switch occurs while external code runs, or that consistency is guaranteed on systems that have truly concurrent threads. Specifically the heap has to be in a consistent state, i.e. that descriptors are not corrupted by thread interactions if another thread triggers a garbage collection. A foreign-function interface should make it as easy and transparent as possible to avoid such situations.

1.3 Classification

The main challenge for a foreign-function interface in the context of high-level programming languages is the interaction with the garbage collector. The interface has to provide a setup so that the external data can live in harmony with the garbage collector of the high-level language. For a programmer, this is the most noticeable difference between the several approaches. Therefore, the existing foreign-function interfaces I investigate in this paper are all classified according to their integration with the garbage collector as their key property. In fact, other aspects like calling conventions and marshalling often fall into place easily, once the interaction with the garbage collector is defined. As a consequence, the main focus of this paper is to describe the garbage-collection related parts of the covered foreign-function interfaces.

Below, I mostly use Scheme as the high-level functional language and C as the external language.

1.4 Outline

Section 2 explains how conservative garbage collectors interact with foreign-function interfaces. Section 3 describes the widely-used GCPROTECT foreign-function interface. Section 4 gives an overview about foreign-function interfaces that use the stable-pointer approach. Section 5 explains in detail the reference-object approach, a JNI-style foreign-function interface. Section 6 briefly describes Scheme 48's new foreign-function interface. Section 7 lists related work and section 8 concludes.

2. Conservative garbage collection

Old versions of PLT Scheme[1] come with a *conservative garbage collector* [6]. Newer versions still give the user the choice to run PLT Scheme with its conservative garbage collector. Bigloo [17] and Guile [20] are Scheme implementations that use conservative

[1] up to PLT Scheme version 360

garbage collectors, as well. Conservative garbage collectors provide automatic memory management without any cooperation from the compiler or the run-time environment of a high-level language. In contrast to precise garbage collectors that can correctly identify all descriptors with the help of the system, conservative collectors cannot exactly differentiate between descriptor and non-descriptor values. Therefore, a conservative collector has to play it safe: It has to treat all memory contents that look like descriptors as descriptors and keep the referenced memory live. The collector searches all data structures that the program uses: It builds the root set by searching for all potential descriptors in the registers and in the stack frames. Then, it follows all possible descriptors that represent heap values from the root set transitively to other objects in the allocated memory and scans the objects for more potential descriptors. It does not reclaim memory that a potential descriptor refers to, thus accidental fake descriptors may cause memory overhead by keeping too much memory live [9].

Since conservative garbage collectors do not need the support of a compiler, they even work with low-level languages like C or any other low-level language that might be used as an external foreign language. Boehm's conservative garbage collector for C and C++ [5], for example, equips programs that are written in either C or C++ with a conservative collector. PLT Scheme uses the Boehm collector.

A foreign-function interface for a high-level programming language can benefit from conservative garbage collection, so does the conservatively-collected foreign-function interface of PLT Scheme: Instead of actively registering descriptors, it has to do nothing and just let the conservative collector take care of finding all descriptors itself. As figure 2 shows, the high-level language (Scheme, for example) shares the objects with the external language.

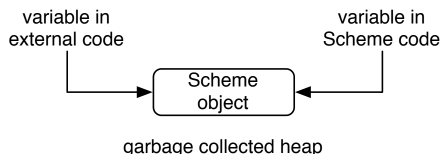

Figure 2. Scheme object under conservative garbage collection

As an example of external code in a system with conservative garbage collection, consider the following implementation of `cons` written in C for some Scheme implementation:

```
scheme_descriptor
cons (scheme_descriptor v1, scheme_descriptor v2)
{
  scheme_descriptor pair;
  pair = allocate (SCHEME_TYPE_PAIR, 2);
  SET_CAR (pair, v1);
  SET_CDR (pair, v2);
  return pair;
}
```

The code is straightforward and gets along without explicit registration of the descriptors with garbage collector: It allocates enough space for the pair and sets the pair's components to the arguments. At any time, the garbage collector can find all Scheme descriptors: The parameters `v1` and `v2` are on the stack, as well as the local variable `pair`.

2.1 Memory overhead

Due to the conservative nature of the garbage-collection algorithm, some random blocks in memory might be kept live longer than necessary due to misidentified descriptors. This may lead to memory leaks. This is a costly price to pay for not having to deal with explicit coordination with the garbage collector.

Additionally, conservative garbage collection comes with other drawbacks: Generally, every kind of automatic memory management poses a run-time cost on the system. Simple collection algorithms pause the execution of the program so that they can perform their collection. Modern collection algorithms primarily try to shorten pause times to make programs more reactive, for example by only covering a part of all used memory in one collection. These collection algorithms are categorized as *incremental collectors*. It is possible, though difficult, to write an incremental conservative collector [9].

Another drawback is even more severe: Since conservative collectors cannot be certain that any value that is visible to the external code is a descriptor, they cannot risk modifying program data and therefore they are constrained to use a non-moving collector for data that is directly accessible from external code. However, Bartlett's *mostly-copying* conservative collector [1] can move data that only has direct pointers from other heap objects, though. But still, it is difficult to manage free memory, because the memory of a running program becomes fragmented: Reclamation of objects leads to holes within the allocated memory, called *fragmentation*. The accumulation of small regions of free storage that are too small to be useful for allocation, even though the sum of free space may be more than sufficient.

2.2 Thread safety

Conservative garbage collection is thread-safe. Collections can occur at any time without special synchronization of threads.

2.3 Usability

A foreign-function interface that is backed by a conservative garbage collector is very easy to use: The programmer does not need to explicitly register objects that are subject to collection with the garbage collector. The conservative collector keeps all accessible objects—and some more—live.

2.4 Portability

The identification of registers, the stack, and other areas that might contain descriptors is highly machine-specific. To get a implementation to work on a variety of machines, the implementation has to provide enough system-specific code so that the conservative garbage collector works on every platform.

3. GCPROTECT

Several implementations of functional programming languages use the *GCPROTECT* mechanism in external code to interface with the garbage collector. For example, variants of this mechanism can be found in the LISP implementation XEmacs [24], the ML dialect Objective Caml [13], and the Scheme implementations Elk [12], Scheme 48 [10], scsh [18], as well as in PLT Scheme's static foreign interface [6]. It is also the mechanism that the authors of the withdrawn SRFI 50 [11] proposed. With this mechanism, the foreign-function interface moves the full liability of communicating to the garbage collector what objects are in use completely to the external code.

The basic idea is that the external code registers all objects that it has in use with the garbage collector. More precisely, all variables of the external code that are bound to Scheme descriptors need to be registered with the garbage collector so that the garbage collector knows the location of all externally held descriptors. The collector will keep the Scheme descriptors in these locations live and update their descriptors if needed. Implementations usually call this registration process *protecting*. Figure 3 shows a protected variable that holds a descriptor in external code.

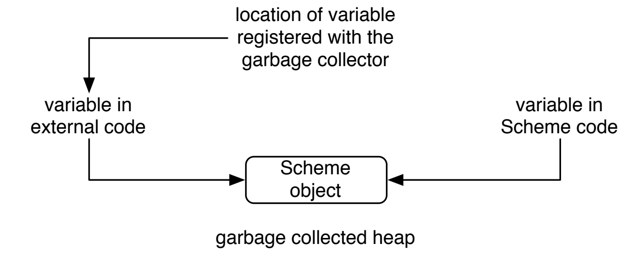

Figure 3. A protected variable

The exact syntax for protection varies from implementation to implementation, but most of the implementations mentioned above use three C macros that go in general like this:

`DECLARE_GCPROTECT` initializes the registry for this function or block. It must be placed in the declarations at the beginning of the enclosing function or block. Usually, it accepts an integer argument that denotes the number of local variables to prepare the registry for.

`GCPROTECT` registers a variable with the garbage collector. The usage of this macro must be within the same scope as `DECLARE_GCPROTECT` and must be before any code that can cause a garbage collection. The variable must contain a descriptor at all times, i.e. the programmer must initialize the variable (usually with the descriptor for the Scheme value `#f`).

`GCUNPROTECT` removes the block's protected variables and unregisters them from the garbage collector. It must be called at the end of the block after any code that may cause a garbage collection.

The programmer adds the above macros to the beginning and end of every C function that accepts or returns Scheme descriptors and lists all variables that may hold a Scheme descriptor, including the function's parameters. All the marshalling happens on the C side as the implementations export functionality to the C code that allows to access and modify all Scheme descriptors.

The following code snippet shows how `cons` could be implemented as an C function with the GCPROTECT mechanism:

```
scheme_descriptor
cons (scheme_descriptor v1, scheme_descriptor v2)
{
  scheme_descriptor pair;
  DECLARE_GCPROTECT (2);
  GCPROTECT (v1);
  GCPROTECT (v2);

  pair = allocate (SCHEME_TYPE_PAIR, 2);
  SET_CAR (pair, v1);
  SET_CDR (pair, v2);

  GCUNPROTECT ();
  return pair;
}
```

Here, the callee protects its call parameters `v1` and `v2`.[2] The programmer needs to protect both variables since the call to `allocate` may trigger a garbage collection and both variables are live across this call. The above code works correctly under the assumption that `SET_CAR` and `SET_CDR` cannot trigger garbage collections. If they could, then the programmer would have to protect the variable `pair` as well.

The above macros work for local variables. Additionally, the foreign-function interface provides primitives to protect and unprotect Scheme objects that outlive a function call, for example because the programmer keeps them as global variables or stores them in external objects whose lifetime is not bound to the current call. There again, the programmer is fully responsible for explicitly protecting and unprotecting the Scheme descriptors.

3.1 Memory overhead

The memory cost of this mechanism is low since an array of protected variables that is shared between the collector and the external code is sufficient for bookkeeping. Variables that the external code does not use do not need to be protected and are thus not kept live by the garbage collector. If the programmer gets it right, the GCPROTECT mechanism does not keep unneeded Scheme objects live.

Globally protected objects are live until the programmer unprotects them explicitly. It is completely up to the programmer to protect the object as long the external code reaches it and to unprotect it as soon as it is no longer in use, like in any language that does not come with an garbage collector. Little overhead is caused by the data structure that keeps track of globally protected external locations.

3.2 Thread safety

Thread safety is difficult to achieve with a GCPROTECT interface: A thread can be interrupted at the moment the external code was going to write a descriptor to a protected location. If a garbage collection occurs before the thread resumes and the descriptor gets written, the garbage collector cannot see this descriptor. The collector may free the object the descriptor points to, or may move it to another address in memory without updating the descriptor. Once the thread resumes, the now broken descriptor gets written to the protected location. This leads to a dangling descriptor in the external code.

Therefore, external code has to either forbid garbage collection in critical sections or set safe points if the code is in a state where it is safe to have a garbage collection. A garbage collection then only runs, when all threads allow garbage collection. All the external code has to be annotated accordingly, which is both a difficult and error-prone task.

3.3 Usability

The mechanism is difficult and error-prone to use, with severe consequences if not used correctly: The programmer has to know under what circumstances the garbage collector may or may not run. If the program forgets to protect a variable, the garbage collector may relocate or free a Scheme object and invalidate the descriptor that the external code uses. If the programmer forgets to unprotect a variable, the garbage collector will treat it as live for the remainder of the program run which may cause a memory leak. However, the foreign-function interface can be equipped to detect missing `GCUNPROTECT`s and do the unprotection for the programmer: For example, the foreign-function interface can record which variables the programmer has protected, which variables the programmer has unprotected and thus derive which variables the programmer forgot to unprotect.

The programming languages' documentations often list a number of rules that the programmer should stick to. The Objective Caml documentation provides four rules [13], the XEmacs Internals Manual [24] lists 13 rules. This is obviously not a user-friendly mechanism. Moreover, these rules are often tightly coupled to the

[2] Other implementations put the caller in charge of protecting the arguments it uses in a subsequent call (this only works if the collector does not move objects).

implementation of the system's garbage collector as they try to list situations when a garbage collection might happen and suggest special treatment in these situations. This imposes a burden on the implementor of the garbage collector and makes changes to the garbage-collection algorithm difficult.

3.4 Portability

The fact that many systems use the GCPROTECT scheme shows that it is a portable approach. It poses some requirements on the language implementation, though:

- Marshalling has to happen on the external side.
- Situations that can trigger a garbage collection have to be well-defined and must not change when the garbage collector algorithm changes.

4. Stable pointers

Stable pointers are an instrument to avoid the need to register descriptors with the garbage collector. Instead of keeping track of garbage-collected objects, this approach only passes references that are not subject to garbage collection to the external code. Variants of this idea can be found in PLT Scheme's dynamic foreign interface [2], the Glasgow Haskell Compiler [21], and the LISP implementations GNU CLISP [8] and Allegro CL [7].

In practice, this can be achieved in two ways:

- External code only works on copies of the data
- High-level code allocates memory manually

I describe both in the following.

External code only works on copies of the data: The basic idea of this variant is that external code must not access objects through descriptors on the heap of the high-level language, but rather works on copies or specially allocated versions of the objects. Figure 4 outlines the setup in Scheme. The programmer is in charge of preparing these copies prior to every external call and cleaning up afterwards. The programmer has to copy the objects to explicitly allocated memory that is not under the control of the garbage collector, so that this memory area and the pointers to it are stable. Then, external code works on the copy of the objects and does not have to worry about the garbage collector freeing or moving the data. After the external call returns, it is again the duty of the programmer to copy the objects or selectively write parts back to the garbage-collected heap of the high-level language that the external code modified and copy the return value of the external code as well.

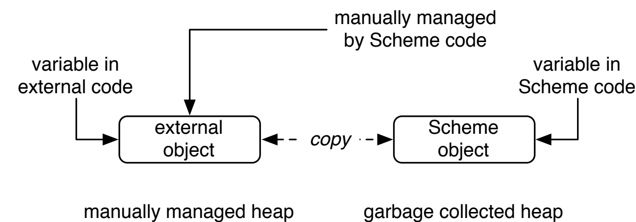

Figure 4. Stable pointers as copies

High-level code allocates memory manually: The high-level language provides an operation that allocates memory that is not under observation by the garbage collector for descriptors that may be used in external code, see figure 5. Since this memory is not automatically managed, it is the responsibility of the programmer to free it after it is no longer used by neither the internal nor the external code. Basically, a foreign-function interface like this lifts the error-prone explicit memory management from lower-level languages into the high-level language. Additional care has to be taken if explicitly allocated storage contains descriptors of automatically managed storage: The garbage collector has to know about the locations of these descriptors to keep them live and up-to-date. Implementations forbid descriptors in manually managed memory or they expect the programmer to provide the needed information to the garbage collector, i.e. with a map of contained descriptor offsets [2].

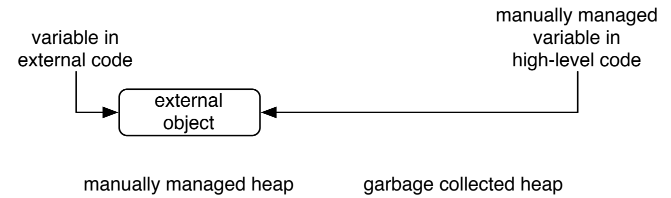

Figure 5. Stable pointers as manually managed memory

Both above-mentioned variants are similar: They put the burden of the manual memory management on the programmer of the high-level code. The only difference is the memory overhead, if two copies of essentially the same data is kept around or if there is only one (explicitly allocated) version. Therefore, I restrict the discussion in the remainder of this section to the copying variant.

Implementations that use this variant do not just copy the descriptors to the external values prior to external calls. In addition, they use this step to marshall from descriptors to the external values, as well. Since this all happens on the high-level side, the high-level language has to know about all the data types of the external code. For C as the external language, the high-level language has to provide a notion for pointers, structs, and all the other C types.

To illustrate this, the following CLISP code [8] shows the usage of ANSI C's `div` function. First, on ANSI C systems, `<stdlib.h>` contains the following declarations:

```
typedef struct {
  int quot;   /* Quotient */
  int rem;    /* Remainder */
} div_t;
extern div_t div (int numer, int denom);
```

To use the external function `div` in CLISP, the programmer lets the LISP system know about the resulting struct type:

```
(def-c-struct (div_t :typedef)
  (quot int)
  (rem int))
```

And then enters the external function as a LISP function:

```
(def-call-out div
  (:arguments (numer int) (denom int))
  (:return-type div_t))
```

To evaluate the call `(div 20 3)`, the foreign-function interface translates the arguments into their C representation, in this example it copies every argument from a LISP number to a C `int`. The C side allocates the struct for the function's return value, sets the struct's components, and returns this struct to LISP. LISP sees the same struct as a `c-struct`:

```
#S(DIV_T :QUOT 6 :REM 2)
```

CLISP provides accessors to the `c-structs` to do unmarshalling into LISP descriptors, i.e. to translate C `ints` to LISP numbers.

4.1 Memory overhead

The memory overhead of this mechanism is significant: Data used in the external code has to be copied to memory regions that the garbage collector does not touch. Thus, this data resides at two locations in memory: On the heap of the high-level side and explicitly malloc'd for the external code. This leads to up to twice the memory consumption for shared objects. Once the external code runs, there is no way to free up no longer needed memory. The explicit memory cleanup cannot happen until the call has returned to the high-level side.

To avoid the memory doubling, Franz's LISP implementation [7] offers an alternative for copying the LISP descriptors to a special memory location: It gives the programmer the ability to register the LISP descriptors that are subject to garbage collection in a table. The table lives on the LISP side and contains the current valid memory location of each registered descriptor. External code can look up a table entry to get the currently valid memory location. Since this memory location is only valid until the next garbage collection occurs, external code has to repeat the look up every time it accesses the LISP descriptor. This is both inefficient and error-prone.

Because of the copying, it is not desirable to use large and complex data structures as arguments for foreign functions. Instead, it is more efficient for a programmer to have as much code that deals with walking the complex data structures on the high-level side and then pass much simpler objects to the external functions.

4.2 Thread safety

Since external code does not directly work on objects that are subject to garbage collection, this approach is thread-safe. External code only works on copies or on objects that are not subject to garbage collection. Thus, thread interactions cannot corrupt data.

4.3 Usability

The foremost advantage of this mechanism is the fact that it does not put any restrictions on the external code. Since all conversions and preparations happens on the inside, the external code does not need to do any kind of bookkeeping. Barzilay [2] uses this mechanism in PLT's dynamic foreign interface to call external functions that are dynamically loaded from compiled external libraries. Since this approach does not require any external glue code or changes to the external code nor recompilation of the libraries, it is truly dynamic. And it makes it easy for a programmer to use external functionality, especially if she prefers to stay in the Scheme world.

On the other hand, this mechanism requires the programmer to manage memory explicitly in the high-level language, and remember to re-enter the results of the external call into the garbage-collected heap. Both are error-prone. In addition, the programmer has to replicate all the external compound values in Scheme that are needed to interact with the external code. The implementations provide various forms of wrappers, helper functions, and macros that aim at making the common cases easy to use and hide the details from the programmer.

Note that the stable-pointer approach only can call into an existing library if it provides access to its functionality dynamically, e.g. only through C functions. If the API of the external library contains macros, the programmer has yet to write and compile external code, because C macros are only part of the library's static interface. Often, a library's API documentation does not distinguish between the macro and truly dynamic parts of the interface, which makes it hard for the user of the dynamic foreign interface.

4.4 Portability

It is not possible to dynamically load compiled libraries in a really portable manner. Object code is different on each platform and naturally contains machine-dependent parts. Additionally, not all compilers make it easy for such a dynamic interface to read the information it needs out of the binary: Due to compiler optimization, for example, it might be different on every platform to calculate the offsets needed to access fields in structs since addresses might be moved off word-boundaries, or might be compressed and thus garbled in any way.

PLT achieves some portability by using `libffi`, a library to connect to `gcc`-compiled libraries, in a slightly adapted version so that it works with `gcc` and Microsoft's compiler [2].

5. Reference objects

Jim Blandy's Scheme implementation Minor [3] uses a foreign-function interface that is quite similar to Sun's Java Native Interface (JNI) [19]. With this mechanism, the foreign-function interface takes care of communicating to the garbage collector what objects it uses in most situations. It relieves the programmer from having to think about garbage collector interactions in the common case.

The foreign-function interface does not give external code direct access to Scheme objects. It introduces one level of indirection as external code never accepts or returns descriptors to Scheme objects. Instead, external code accepts or returns *reference objects* that refer to Scheme objects, see figure 6.

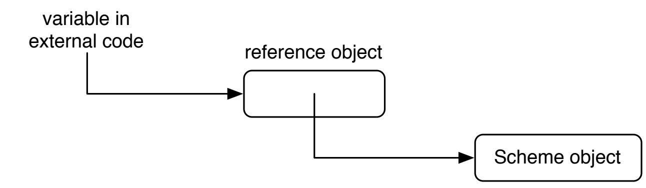

Figure 6. Reference objects

This indirection is only needed as an interface to external code, interior pointers in Scheme objects are unaffected. The interface provides functionality to the external code that allows to access reference objects' Scheme descriptors and does the marshalling. All the marshalling happens on the external side.

Minor provides two kinds of reference objects:

local references A local reference is valid for the duration of a function call from the high-level language to external code and is automatically freed after the external function returns to the virtual machine.

global references A global reference remains valid until external code explicitly frees it.

Scheme objects that are passed to external functions are passed as local references. External functions return Scheme objects as local references. External code has to manually manage Scheme objects that outlive a function call as global references. Scheme objects outlive a function call if they are assigned to a global variable of the external code or stored in long-living external objects. A common example are Scheme functions that are used as call-backs for an external GUI library.

A local reference is valid only within the dynamic context of the native method that creates it. Therefore, a local reference behaves exactly like a local variable in the external code: It is live as long as external code can access it. To achieve this, every external function in the interface that accepts or returns reference objects takes a *call object* as its first argument. The call object holds all the references

that belong to a call (like the call's arguments and return value) to external code from Scheme. External code may pass a local reference through multiple external functions. The foreign-function interface automatically frees all the local references a call object owns, along with the call object itself, when an external call returns to Scheme.

This means that in the common case of Scheme calling an external function that does some work on its arguments and returns without stashing any Scheme objects in global variables or global data structures, the external code does not need to do any bookkeeping, since all the reference objects the external code accumulates are local references [4]. Once the call returns, the foreign-function interface frees all the local references.

As an example, the following code shows how an external `cons` could be implemented in C:

```
ref_t
cons (call_t call, ref_t v1, ref_t v2)
{
  ref_t pair;
  pair = allocate (call, SCHEME_TYPE_PAIR, 2);
  set_car (call, pair, v1);
  set_cdr (call, pair, v2);
  return pair;
}
```

In addition to the car and the cdr, `cons` takes a call object. A function call from Scheme to the above function `cons` associates the parameters `v1` and `v2` with the call object. The `allocate` function associates the reference object that is bound to the variable `pair` with the call argument as well. It does so by wrapping the Scheme descriptor it allocates in a reference object. Thus, all Scheme descriptors to which the external code has access are known to the garbage collector, without any explicit form of registration. Whenever a collection occurs, these objects will be kept live and their descriptor remains up-to-date.

In the rare cases where external code needs a reference object to survive the current call, the external code needs to do explicit bookkeeping. Local references must not be stored in global variables of the external code or passed to other threads, since all local references are freed once the call returns, which leads to a dangling pointer in the global variable or other thread respectively. Instead, a programmer can promote a local reference to a global reference that she can safely store in a global variable or pass to another thread as global references will survive the current call [14]. Since the foreign-function interface never automatically frees global references, the programmer must free them at the right time. Managing global references means more work, but this is the kind of work that a C programmer is used to anyway: Like any other kind of object that the programmer allocates explicitly (or promotes to a global reference, in the terms of the foreign-function interface), a global reference needs to be freed explicitly as well. The same holds for sharing a object between multiple threads: The programmer has to ensure that one thread is not using a global reference while another thread is freeing it [4].

5.1 Memory overhead

Each reference object consumes a certain amount of memory itself, in addition to the memory taken by the referred Scheme object itself. Even though local references are eventually freed on return of an external call, there are some situations where it is desirable to free local references explicitly, since waiting until the call returns may be too long or never happen, which could keep unneeded objects live:

- External code may create a large number of local references in a single external call. An example is the traversal of a list: Each call from external code to the functions that correspond to `car` and `cdr` returns a fresh local reference. To avoid the consumption of storage for local references proportional to the length of the list, the traversal must free the no-longer-needed references as it goes [4].
- The external call does not return at all. If the external function enters an infinite event dispatch loop, for example, it is crucial that the programmer releases local references manually that he created inside the loop so that they do not accumulate indefinitely and lead to a memory leak [14].
- External code may hold a local reference to a large Scheme object. After the external code is done working on this object, it performs some additional computation before returning to the caller. The local reference to the large object prevents the object from being garbage collected until the external function returns, even if the object is no longer in use for the remainder of the computation. It is more space-efficient if the programmer frees the local reference when the external function does not need it any longer and will not return for quite some time [14].

Additionally, Blandy describes common situations where local references are created solely to be passed to another function, and afterwards never used again [4]. For example, the programmer uses this code snippet to produce a local reference to the pair `(23 . 42)`:

```
ref_t twentythree = enter_integer (call, 23);
ref_t fortytwo = enter_integer (call, 42);
ref_t pair = cons (call, twentythree, fortytwo);
free_local_ref (call, twentythree);
free_local_ref (call, fortytwo);
```

In this example, the programmer frees the arguments of the call to `cons` since both references are no longer needed. As this is a common pattern, Minor's C API offers variants for many functions that free all the references they are passed to as arguments, in addition to whatever else the functions do. The above example can be written more efficiently like this:

```
ref_t pair = ad_cons (call,
                      int_to_number (call, 23),
                      int_to_number (call, 42));
```

The resulting code is more legible and elegant as well as more memory efficient.

To summarize, memory overhead is low in the common case, but there are some situations, that need special attention to keep it low. The situations are well-defined and some of the situations can be seen as opportunities to optimize, i.e. their recognition and treatment is not crucial for the external code to work correctly.

5.2 Thread safety

Since reference objects are not mutable, this mechanism can easily be implemented in a thread-safe manner: Only when a Scheme object is initially wrapped into a reference object, a thread switch or a truly concurrent thread that runs the garbage collector can corrupt the external code's descriptors. Thus, it is sufficient for the foreign interface to treat the creation of reference objects as a critical section where a GC must not move the underlying Scheme object.

5.3 Usability

This foreign-function interface takes a significant burden off the programmer as it handles most common cases automatically. If all

the Scheme objects are live for the extent of the current external call, the programmer does not have to do anything at all. Since the lifetime of the Scheme objects is then identical with the lifetime of the according reference objects. In this case, the systems automatically manages both for the programmer.

For Scheme objects that outlive the current call, things get more involved, as the programmer has to manage their reference objects manually. But the need to manage long-living objects manually is familiar to C programmers.

The fact that the memory that external calls keep live is bound to the extent of an external call entails the need for freeing local references explicitly in certain situations described above. Explicitly freeing local references to keep the memory overhead low for long-lasting external calls is still error-prone, however.

Using this foreign-function interface does not make the code more complex; the code stays compact and readable. The programmer has to get accustomed to passing the call argument around. Again, things get uglier if the situation forces the programmer to explicitly free local references, since then the code gets cluttered with deallocation statements.

5.4 Portability

This approach is easily portable, as it does not impose any restrictions on the virtual machine and the garbage collector. The approach builds a simple layer of abstraction on top of the underlying run-time system.

6. Scheme 48

The upcoming release of Scheme 48 contains a JNI-style foreign-function interface as described in section 5. Out of all the different approaches I covered here, it is the one that is the easiest to use, the most portable, and does not pose any restrictions on the run-time system.

It was easy to implement on top of the Scheme 48 system: The garbage collector of Scheme 48 provides a straightforward interface to add locations to the root set, which the new foreign-function interface uses to register local and global references. Additionally, Scheme 48's old foreign-function interface already supports raising exceptions from external code and takes care of cleaning up the call stack on non-local exits. I was able to re-use the already existing functionality for the new foreign-function interface.

Thus, Scheme 48 currently supports two foreign-function interfaces: The old GCPROTECT-style (see section 3) and the new JNI-style interface live side by side. This is useful for comparing both approaches and essential for backwards compatibility—there are many lines of external code that use the GCPROTECT interface. Although it is easy to rework existing external code from GCPROTECT to JNI-style—change all function definitions and function calls so that they only accept reference objects and return only reference objects, and add a call object to every function—it may take a while until all external code works with the new foreign-function interface. Until now, we have already ported the following external libraries to the new foreign-function interface: POSIX, networking code, SRFI 27, and the Oracle Call Interface for Scheme 48 (oci48). Eventually, I will remove the old GCPROTECT-style interface.

In the already ported libraries with over 4,000 lines of code, there are only two global references left that need to be managed explicitly, all other Scheme values that the libraries use are local references and are thus managed automatically by the foreign-function interface. With the new interface, I was able to remove over 200 lines of GCPROTECT-related code from the libraries.

7. Related work

Another approach to a foreign-function interface that came up on the SRFI-50 discussion list [16] is implemented in Lord's Pika Scheme [15]. Note that Pika is not usable yet and that it is currently not under development. Since the ideas are not proved by demonstration, I just give a quick overview: It is similar to the JNI approach in that it also only works on reference objects, does not put any restriction on the garbage collector, and works nicely with threads. The biggest difference is that reference objects are stack-allocated on the external side and thus get freed upon exit from the lexical block that owns them, which makes it more exact than JNI that needs the call to return. The external code written in Pika style is more verbose than external code that uses JNI style, though:

1. Functions may only return objects by reference, they may not provide them as normal return values.
2. Variable declarations need to be enclosed in structs.
3. These structs need to be registered and unregistered with the garbage collector.

While the first two points are arguably a unusual way to write C code, the last point describes a mechanism that is related to the GCPROTECT mechanism, see section 3.

8. Conclusions

Memory management is hard. Memory management across multiple programming languages is even harder. Thus, the most important issue that a implementor has to consider in designing a foreign-function interface to make functionality available across different programming languages is the management of memory. I reviewed different approaches to foreign-function interfaces that are used in implementations of functional programming languages:

Conservative garbage collection (see section 2): Foreign-function interfaces that are backed by a conservative garbage collector are becoming rare. While some implementations like PLT Scheme moved away from a conservative collector, other implementations have always had exact garbage collectors. Although such a foreign interface is very easy to use by a programmer and it is thread-safe, the disadvantages outweigh the simplicity of writing external code.

GCPROTECT (see section 3): The dissatisfaction with the GCPROTECT mechanism and the difficulty of writing external code with it was the reason to undertake this survey in the first place. The GCPROTECT scheme is error-prone since the programmer has to manage local and global variables explicitly and it is difficult to track down errors.

Stable pointers (see section 4): A foreign-function interface that allows the use of dynamically loaded external libraries without the need to write a single line of external code is appealing to programmers. However, the code that the programmer has to write to access the external functionality and to do object marshalling has to contain low-level details and brings explicit memory management into the high-level world. The external code gets lifted to high-level syntax. Additionally, this approach is not really portable since it requires nonstandard extension for the C compiler.

Reference objects (see section 5): The reference-object approach is easy to use, is fully portable, can be easily made thread-safe, and does not pose any restrictions on the run-time system. The programmer only has to manage those objects explicitly that outlive an external call, the foreign function interface manages objects that do not outlive the call automatically. In addition, it is easy to implement and add to an existing implementation

of a high-level programming language as it is a simple layer of abstraction on top of the underlying run-time system.

Even if the described interfaces to external code try to make it as easy as possible for the programmer, it is never realistic to expect to be able to completely avoid explicit memory management in C.

Acknowledgments

I thank Mike Sperber and Eric Knauel for proof-reading and their valuable suggestions, Harald Glab-Plhak for testing Scheme 48's new foreign-function interface, and Jim Blandy for bringing a JNI-style foreign-function interface into the Scheme world. Suggestions by the anonymous reviewers led to improvements in the final version of this paper.

References

[1] J. F. Bartlett. *Mostly-Copying garbage collection picks up generations and C++*. Technical Note, DEC Western Research Laboratory, Palo Alto, CA, 1989.

[2] E. Barzilay and D. Orlovsky. Foreign interface for PLT Scheme. In *Proceedings of the Fifth ACM SIGPLAN Workshop on Scheme and Functional Programming*, pages 63–74, 2004.

[3] J. Blandy. *Minor Scheme*. 2007. `http://www.red-bean.com/trac/minor/`.

[4] J. Blandy. *Minor's main header for the C API*. 2007. `http://www.red-bean.com/trac/minor/browser/trunk/include/minor/minor.h`.

[5] H.-J. Boehm. *A garbage collector for C and C++*. 2007. `http://www.hpl.hp.com/personal/Hans_Boehm/gc/index.html`.

[6] M. Flatt. *Inside PLT MzScheme*. Part of the PLT Scheme distribution, 2007. `http://download.plt-scheme.org/doc/372/html/insidemz/insidemz.html`.

[7] Franz Lisp. *Foreign function interface*. 2008. `http://www.franz.com/support/documentation/8.1/doc/foreign-functions.htm`.

[8] B. Haible, M. Stoll, and S. Steingold. *Implementation Notes for GNU CLISP*. Part of the GNU CLISP distribution, 2008. `http://clisp.cons.org/impnotes`.

[9] R. Jones and R. Lins. *Garbage Collection Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons Ltd., 1996.

[10] R. Kelsey, J. Rees, and M. Sperber. *The Incomplete Scheme 48 Reference Manual for release 1.8*. Part of the Scheme 48 distribution, 2008. `http://s48.org/1.8/manual/manual.html`.

[11] R. Kelsey and M. Sperber. *SRFI 50: Mixing Scheme and C*. 2003. `http://srfi.schemers.org/srfi-50/srfi-50.html`.

[12] O. Laumann. *Building Extensible Applications with Elk – C/C++ Programmer's Manual*. 1995. `http://www-rn.informatik.uni-bremen.de/software/elk/doc/cprog.html`.

[13] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. *The Objective Caml system*. 2007. `http://caml.inria.fr/pub/docs/manual-ocaml`.

[14] S. Liang. *Java Native Interface: Programmer's Guide and Specification*. 1999. `http://java.sun.com/docs/books/jni/`.

[15] T. Lord. *Pika Scheme*. 2003. `http://regexps.srparish.net/www/#pika`.

[16] Members of the SRFI community. *SRFI 50 Pre-Withdrawal Discussion Archive*. 2003. `http://srfi.schemers.org/srfi-50/mail-archive/maillist.html`.

[17] M. Serrano. *Bigloo Scheme*. 2008. `http://www-sop.inria.fr/mimosa/fp/Bigloo/bigloo.html`.

[18] O. Shivers, B. D. Carlstrom, M. Gasbichler, and M. Sperber. *Scsh Reference Manual*. Part of the scsh distribution, 2005. `http://www.scsh.net/docu/html/man.html`.

[19] Sun Microsystems. *Java Native Interface Specification*. 2003. `http://java.sun.com/j2se/1.5.0/docs/guide/jni/spec/jniTOC.html`.

[20] The Free Software Foundation. *GNU Guile*. 2008. `http://www.gnu.org/software/guile`.

[21] The GHC Team. *The Glorious Glasgow Haskell Compilation System User's Guide*. Part of the GHC distribution, 2007. `http://haskell.cs.yale.edu/ghc/docs/6.8.2/users_guide.pdf`.

[22] The GTK+ Team. *The GTK+ Project*. 2008. `http://www.gtk.org`.

[23] R. Urban. *Design issues for foreign function interfaces*. 2004. `http://autocad.xarch.at/lisp/ffis.html`.

[24] B. Wing, S. Turnbull, M. Buchholz, H. Niksic, M. Neubauer, O. Galibert, and A. Piper. *XEmacs Internals Manual*. Part of the XEmacs distribution. `http://www.xemacs.org`.