Malloc Pointers and Stable Pointers: Improving Haskell's Foreign Language Interface

Alastair Reid*
Computing Science Department, University of Glasgow
Glasgow G12 8QQ, Scotland

September 26, 1994

Abstract

The Glasgow Haskell compiler provides a foreign language interface which allows Haskell programs to call arbitrary C functions. This has been used both to implement the standard Haskell IO system and a variety of applications including an arcade game [8], and a graphical user interface to a database [19].

The theoretical problems associated with using impure functions from pure functional languages are avoided through the use of monads [17]; and the mismatch between strict languages with no garbage collection and lazy languages with garbage collection is tackled by unboxing (that is, forcing evaluation of arguments and stripping off any header information) [15].

Whilst this works well for simple examples, it is unsuitable when one wants to pass arguments (or results) which are lazy, polymorphic or very large. We describe two extensions to the garbage collector which solve these problems by allowing better interaction between the Haskell garbage collector and memory allocation in the imperative world.

1 Introduction

The LISP and Standard ML communities have known for some time that impure functional languages are useful for more than just symbolic manipulation and toy programs — functional programs can also provide sophisticated user interfaces [9, 7] and can be used for systems programming tasks such as implementing communications protocols [10].

The (lazy) pure functional world is beginning to demonstrate that pure functional languages are also good for writing programs traditionally considered to be outside their domain. A good example is Carlson and Hallgren's Fudget system [5] — an efficient library of "functional widgets" which can be used to implement graphical user interfaces.

*email: areid@uk.ac.glasgow.dcs; http: //www.dcs.gla.ac.uk/~areid

It is therefore *technically* possible to discard most (or all?) existing imperative libraries and reimplement them all in a (pure) functional language but there are compelling reasons why we cannot or should not do so:

Development Effort In addition to the libraries that come supplied with operating systems and compilers, there are several substantial libraries of freely-available software such as the X widget system, and the Free Software Foundation's libraries.

If each such library must be reimplemented in a functional language before it can be used, functional programs will require considerably more effort to produce than imperative programs.

Performance Despite significant improvements in compiler technology, highly tuned imperative programs remain significantly more efficient in time and (especially) space than equivalent lazy functional programs.

There are two major problems in calling imperative library routines from a pure, lazy functional language:

Control flow In a pure functional language, the order of evaluation can only affect termination and resource usage — this leaves the optimisation phases of the compiler and the runtime system considerable freedom to choose an evaluation order. If "impurities" (i.e. side effects) are introduced, steps must be taken to constrain the choice of evaluation order.

Passing Data The older (and most popular) imperative languages (such as `C` and `Pascal`) either allocate objects at a fixed address, on the stack or on the heap according to their lifetime. All heap allocated objects must be explicitly deallocated.

In functional languages most objects are heap allocated and automatically deallocated through garbage collection. (A lifetime analysis is also possible — allowing the compiler to decides where to allocate an object or to perform "compile-time" garbage collection.)

Issues of control flow can be dealt with by writing programs in continuation passing style or monadic style [17]; therefore this paper is primarily concerned with issues associated with passing data to and from an imperative language.

Section 2 describes the Glasgow Haskell Compiler's foreign language interface. Section 3 describes problems in passing arguments that are: polymorphic; lazy; large; persistent from one call to another; or functions. We propose two new types and associated operations to overcome these problems. These extensions are included in the latest release of the Glasgow compiler — the implementation is outlined in section 4.

2 The Raw Iron

This section outlines some of the primitive facilities for calling imperative functions provided by `ghc`: the "raw iron" from which higher level facilities are built. Further facilities are described in section 3 when we discuss the extensions we made to the compiler.)

2.1 C calls and the `PrimIO` monad

For our purposes, the most important of `ghc`'s primitive operations is `_ccall_` which is used to call an arbitrary C function from within Haskell. To call the standard C trigonometric function `sin`, one writes

```
_ccall_ sin (3.2::Double)
```

and to call the standard C output function `printf`, one writes

```
_ccall_ printf "The answer is %d.\n" (42::Int)
```

Note that the compiler must be able to determine the type of the arguments (and results) of a `_ccall_` — hence the explicit type signatures on arguments that would otherwise be ambiguous.

Note too that `_ccall_` does not distinguish between pure functions (such as `sin`) and impure functions (such as `printf`) — both are assumed to have side-effects. The theoretical and practical problems arising from calling impure functions from a pure functional language are avoided by using a monad to force strict sequencing of `_ccall_`s. We refer the reader to [17, 14, 16] for a detailed discussion of the monadic approach but note that the type `PrimIO` and functions `thenPrimIO`, `returnPrimIO` and `unsafePerformPrimIO` were called `IO`, `thenIO`, `returnIO` and `performIO` respectively in [17].

If a programmer believes that a computation of type `PrimIO` α is referentially transparent and does not have any side-effects, `unsafePerformPrimIO` :: `PrimIO` $\alpha \rightarrow \alpha$ can be used to eliminate the need to include it the main thread of execution.[1]

For example, the following program assumes the existence of two imperative functions `readInt` and `writeInt` to read two numbers and write their sum.

```
mainPrimIO =
  _ccall_ readInt                       `thenPrimIO` \ x ->
  _ccall_ readInt                       `thenPrimIO` \ y ->
  _ccall_ writeInt ((x+y) :: Int)       `thenPrimIO` \ () ->
  returnPrimIO ()
```

(Haskell Notation: If `f` is a binary function, the function `` `f` `` is a binary infix operator. The notation `\x -> e` represents the lambda term $\lambda x.e$; the scope of x extends as far to the right as possible.)

2.2 `CCallable` and `CReturnable` data types

The representation of values used in a language implementation depends on whether automatic garbage collection is provided or not: languages with garbage collection typically add a header to the front of every object for use during garbage collection. This header must be removed from arguments to a `_ccall_` and added to the front of results. In addition, values in lazy languages may be represented by a "thunk" representing the calculation required to calculate

[1] The word "unsafe" is intended to remind the programmer that the compiler has no way of checking whether or not the operation has side-effects. Apart from the obvious debugging uses, we have yet to see a program that *safely* uses this function to call a side-effecting function.

that value. In most cases, imperative functions must be passed the value of a "thunk" rather than the "thunk" itself.

The necessary conversions could be made explicit through the use of "unboxed types" [15] but `ghc` automatically performs these conversions on the following basic data types: `Char`, `Int`, `Float` and `Double`. These types can be both passed as arguments in `_ccall_`s and returned as results from `_ccall_`s — we call such types `CCallable` and `CReturnable` respectively. (There is a mechanism for expanding the set of `CCallable` and `CReturnable` types which we shall not elaborate here.)

2.3 Nonstandard Types

In addition to these standard Haskell types, `ghc` provides the non-standard types `_Word` (an unsigned integer), `_Addr` (a machine address), `_ByteArray` (a contiguous region of bytes in the Haskell heap which may be read but not modified) and `_MutableByteArray` (a contiguous region of bytes in the Haskell heap which may be both read and written). `ghc` provides a small set of operations on these types including equality tests, bit manipulations and array allocation and indexing operations.

`_MutableByteArray`s can be used to return multiple arguments from a C function. For example one could write the following to call the standard C function `sincos` which takes a double-precision angle and the address of two double-precision variables and writes the *sine* and *cosine* of the angle into the two variables.

```
sincos :: Double -> PrimIO (Double, Double)
sincos a = unsafePerformPrimIO (
  newDouble                              `thenPrimIO` \ sv ->
  newDouble                              `thenPrimIO` \ cv ->
  _ccall_ sincos_wrapper a sv cv         `thenPrimIO` \ () ->
  readDouble sv                          `thenPrimIO` \ sin ->
  readDouble cv                          `thenPrimIO` \ cos ->
  returnPrimIO (sin, cos)
  )
```

The non-standard functions `newDouble` and `readDouble` are defined using primitive operations on `_MutableByteArray`s. They are used to allocate enough memory to hold a `Double` and to extract a `Double` from an array. The wrapper function `sincos_wrapper` performs type coercions and, most importantly, overcomes any restrictions on alignment of `double` imposed by modern RISC architectures through the use of some machine-specific macros for assigning unaligned `double`s.

```
void
sincos_wrapper(StgDouble a, StgByteArray *sin, StgByteArray *cos)
{
  double s, c;
  sincos(a, &s, &c);
  ASSIGN_DBL( (StgDouble *) sin, s);
  ASSIGN_DBL( (StgDouble *) cos, c);
}
```

The need to write a small wrapper function for every imperative function called is rather tedious and error-prone. A far more reliable approach would be to write a wrapper-generator which automatically constructed the `Haskell` and `C` wrapper functions from a list of type signatures.

3 Problems with the raw iron

Many values found in Haskell programs are unevaluated heap-allocated objects of arbitrary size such as Lists and arbitrary precision Integers. Most values in imperative programs are fully evaluated stack-allocated objects of a fixed (small) size such as 32-bit integers. This mismatch between Haskell and the languages it is calling can usually be solved by fully evaluating the value and passing it to the imperative function using the standard argument passing convention for that architecture. This standard approach is restricted in a number of ways:

Laziness: Since all arguments are evaluated before calling the `C` function, one cannot write lazy functions. Given that `C` is a strict language, this seems perfectly reasonable. However, O'Donnell [12] describes a hardware implementation of arrays which provides extensible, sparse functional arrays (called "ESF arrays"). By exploiting the parallelism inherent in hardware, O'Donnell is able to perform both update and lookup in constant time. When using such a device from a lazy language, one would obviously require these arrays to contain (pointers to) unevaluated values rather than just integers.

Polymorphism: Since `Haskell`'s evaluation mechanism and `C`'s argument passing mechanisms vary from one type to another, one cannot write functions exhibiting pure polymorphism. Again, since `C` does not provide pure polymorphism, this seems perfectly reasonable. However, problems would arise if one wished to use O'Donnell's ESF arrays to implement arrays of characters, floating point numbers or even arrays.

Large persistent data structures: Since `C`'s argument passing convention restricts one to passing "small" values (ie values that will fit in registers) one cannot directly pass (or return) large objects but must pass their address instead.

If the object being passed has a short well-defined lifetime, it is reasonable to use tricks such as that used in section 2.3 where we explicitly allocated space on the heap, call the function `sincos` and read the values out. Similarily, if returning an array of characters (say) from `C` to `Haskell`, one might explicitly allocate an array on the `C` heap, copy the result into the array, return the address of the result to `Haskell`, read the characters from the array into a list and call the `C` function `free` to explicitly deallocate the array.

As well as being somewhat inefficient, such solutions can cause garbage collection problems if an object "persists" from one function call to the next:

- Many modern garbage collectors move objects during garbage collection. This doesn't normally cause a problem because all pointers to an object are updated when the object is moved. If the garbage collector is unaware that the imperative world has a pointer to a Haskell heap object, the pointer cannot be updated and errors can arise.
- Garbage collection reclaims storage used by objects that are no longer referenced. If the garbage collector is unaware that the imperative world has a reference to a Haskell heap object, it might be deallocated when references still exist.
- Since most `C` implementations do not have automatic garbage collection (see, for example, [3] for an exception), it is necessary to explicitly deallocate heap-allocated objects as the last reference to the object is deleted. This is hard to do correctly in an imperative language; it is virtually impossible in a lazy functional language.

Haskell functions In `C` a function may be represented by a pointer to the corresponding machine code for that function. Such pointers may be passed to and returned from other functions. In `Haskell`, it is not enough to pass a pointer to the machine code, one must also pass any values bound to free variables occuring in the function. Since the number of free variables may be changed by evaluation and by optimisations (both at compile time and during garbage collection), there is no easy way to convert `Haskell` functions to their corresponding representation in `C`.

We have come across several examples where it would be useful to be able to pass `Haskell` functions to `C`:

- When writing graphical user interfaces under X, we must provide the "widgets" with actions ("callbacks") to perform when the user clicks a mouse on a widget or closes a window. If one wishes to write graphical user interfaces in Haskell, the natural way of defining which callback to call in response to a given event is to store the address of the (possibly heap-allocated) callback routine in the Widget — almost exactly as one does for C. (Our earlier paper [19] describes a rather *ad hoc* approach which avoids the need to extend the garbage collector.)
- The Haskell 1.3 IO proposal defines a function

  ```
  setInterrupt :: IO () -> IO (IO ())
  ```

  which allows a monadic program to specify an "interrupt handler" to be executed when a console interrupt occurs.

  An efficient implementation requires the runtime system to store the current "interrupt handler" in a global `C` variable. (Since this is part of the runtime system, it would be possible to build it into the garbage collector as a special case.)

Our solution to the above problems is to add two new primitive types to the language: `_MallocPtr`s are pointers from the `Haskell` heap to objects in the `C` heap; and `_StablePtr`s are pointers from the `C` heap to objects in the

Haskell heap. We modify the garbage collector to take these objects into account during garbage collection.

3.1 Malloc Pointers

In principle an object of type _MallocPtr is just an index into a table of addresses of objects in the C heap. (The name derives from the way most objects in the C heap are allocated: by calling the standard library function malloc. However, we expect _MallocPtrs to be values such as file handles, ESF array identifiers, etc.)

A _MallocPtr is automatically allocated on return from a _ccall_ with result type _MallocPtr and is automatically dereferenced when passed as an argument to _ccall_. When the ghc runtime system detects that a _MallocPtr is no longer accessible, the _MallocPtr is deallocated and a programmer-supplied C function

```
void FreeMallocPtr( StgMallocPtr mp )
```

is called.

Since ghc will only detect that a _MallocPtr is inaccessible during garbage collection, C programs may call the C function StgPerformGarbageCollection to force a garbage collection.[2]

3.2 Stable Pointers

In principle (and in practice), an object of type _StablePtr α is an index into a table of addresses of objects of type α in the Haskell heap. When the Haskell garbage collector moves an object to which there is a _StablePtr, the corresponding entry in the table is updated; an object will not be deallocated if a stable pointer to it exists.

Stable pointers may be passed to and returned from _ccall_s. The following operations may be used from Haskell to explicitly allocate, deallocate and dereference _StablePtrs.

```
makeStablePointer  :: a -> PrimIO (_StablePtr a)
freeStablePointer  :: _StablePtr a -> PrimIO ()
derefStablePointer :: _StablePtr a -> a
```

Note that, because of the use of explicit allocation and deallocation, space leaks can result if stable pointers are not released when finished with. It is for this reason that makeStablePointer is not of type $\alpha \rightarrow$ _StablePtr α: use of the PrimIO monad avoids any risk of a single use being "optimised" into several uses.

There is also a C procedure

```
void FreeStablePtr( StgStablePtr sp )
```

[2] If a _ccall_ is likely to perform a garbage collection, it is necessary to take considerable care that no registers contain live heap pointers and to make the contents of essential registers such as the heap pointer available. Rather than pay this extra cost on every _ccall_, we provide a special form _ccall_GC_ which should be used if StgPerformGarbageCollection can be called by the C function being called. It is a checked runtime error to call StgPerformGarbageCollection during a plain _ccall_.

which frees a stable pointer. (This is the only `ghc`-provided operation which may be called within `FreeMallocPtr`.)

To support the callback mechanism, there are also `C` functions

```
void enterPrimIO ( StgStablePtr sp )
int  enterInt    ( StgStablePtr sp )
...
```

which calls stable pointer of type `StablePtr (PrimIO ())`, `StablePtr Int`, etc. Sadly, there is little possibility of adequately typechecking these calls.

3.3 Applications

Of the two mechanisms, we have found `_MallocPtr`s to be the most useful. In addition to the uses mentioned above, some possible applications include:

- Haskell provides an operation to lazily read a file. The result of this function is a string consisting of the contents of the file but, because the read is performed lazily (ie as each character in the file is demanded), it is still possible to process a file in constant space.

  One problem is in automatically closing these files. If the end of the file is reached, the file may be simply closed; but if the file is discarded before the end of the file is reached, the file will remain open. As well as preventing other programs from writing to the file, this might result in the program running out of file handles (on a UNIX system, each process is only allowed to open a certain number of files).

- In the X window system, each display runs a server which can draw lines, text, etc. on the screen. Programs wishing to perform graphics on a given display connect to the appropriate server and send a stream of requests to draw images. To reduce the amount of network communication, bitmaps, fonts, colours, etc are stored in the server's memory. Allocation and deallocation of these resources is performed explicitly. (One might think that resources could be deallocated when a connection is broken. The X protocol allows resources to be shared between processes but does not require clients to inform it of such sharing — this prevents such deallocation from occuring.)

  A common problem *with imperative programs* is that they fail to deallocate some of the resources allocated to them. This leads to a gradual degradation of performance and functionality as the resources of the server gradually disappear and, to alleviate this problem, most servers are initially allocated far more memory than they require.

  It should be possible to avoid this problem by using `_MallocPtr` to represent all server objects. If any allocation request fails, the client should call `StgPerformGarbageCollection` in the hope of freeing some unreachable `_MallocPtr`s.

  One potential problem with this scheme is that it is possible for one quiescent program (a mail-reader, say) to hog all the resources if it does not perform a garbage collection very often. This could perhaps be overcome by modifying the server to inform *all* clients when it was low on resources.

A program receiving this notification could perform a garbage collection in the hope of freeing some `_MallocPtr`s.

4 Implementation

This section describes how stable pointers and malloc pointers are implemented.

4.1 Malloc Pointers

Programmers are encouraged to think of `_MallocPtr`s as indexes in a table of pointers into the C Heap. However, since the only operation on these "indexes" is to dereference them and to scan through them all, it is possible to implement the "table" as a linked list without any loss of efficiency. (This avoids the need to implement complex operations to resize the table.)

(There is one slightly subtle aspect to this choice of representation: during garbage collection, an object is normally considered to be "live" if it is a "root" or it is pointed to by any other "live" object; it is important not to treat the link to the next `_MallocPtr` in the list as an ordinary pointer or `_MallocPtr`s will only die when all `_MallocPtr`s further up the list die. This might badly affect a conservative garbage collector operating without knowledge of the internal structure of heap objects.)

4.2 Stable Pointers

A `_StablePtr` is represented as an index into a table. The table is allocated on the heap and may be resized to suit demand. Every time the table overflows, it is doubled in size resulting in an amortised constant-time operation. We maintain a "stack" of unused entries in the table. (This stack could be eliminated by threading a free list through the table of "unstable" pointers. This would require a way of distinguishing "unstable pointers" from links in the list.)

Ignoring standard header information (used by the garbage collector) a stable pointer table closure looks like this:

$NPtrs$	SP_0	SP_1	$\ldots$	SP_{n-1}	Top	s_0	s_1	$\ldots$	s_{n-1}

The fields are:

$NPtrs$ The number of (stable) pointers.

SP_i An "unstable" pointer to a closure. This is the pointer that gets updated when the garbage collector moves an object we have a stable pointer to. If the pointer is not in use, it points to a preallocated static closure.

Top The index of the first element above the top of the stack.

s_i An entry in a stack of unused pointers. Entries in use will contain a number in the range $0 \ldots n-1$.

For example, with $n = 4$ and pointers `0` and `3` in use (pointing to `p1` and `p2` respectively), the table might look like this:

4	$p1$	?	?	$p2$	2	1	2	?	?

4.3 Garbage Collection

Glasgow Haskell supports four different garbage collectors:

- a 2-space copying collector [6];
- a 1-space compacting collector [11];
- a "dual mode" collector which operates either as a two space collector or a compacting collector depending on the amount of live data [20]; and
- a generational collector [1, 21]. This collector maintains just two generations: the new generation is collected using a two-space collector; and the old generation is collected using a one-space collector. To allow separate collection of the generations, a list of pointers from the old generation to the new generation is maintained.

The changes required to the collectors are as follows:

4.3.1 Copying collector

2-space collection consists of a single main phase which alternates between copying live heap objects into a new area of memory (evacuation) and scanning copied objects for pointers to uncopied heap objects (scavenging). When an object is evacuated, it is overwritten by a "forwarding pointer" which points to the copy — this allows all references to an object to be updated with the same address.

The stable pointer table is treated as a "root" during garbage collection and collected in the normal way.

After this main phase has completed, elements of the `_MallocPtr` List (which will still be in the from space) is scanned checking which objects have been replaced by forwarding pointers (and so are still live) and which are now dead and can be released. Each live object is added to the new `_MallocPtr` list.

There is one slight subtlety. When a heap-object is overwritten with a forwarding pointer, care must be taken *not* to overwrite the link to the next `MallocPtr` in the chain since this is required later in the garbage collection. (In all other objects, the garbage collector is free to trash the contents of an object as soon as it has been copied.)

4.3.2 Compacting Collector

1-space collection consists of three main phases: a traditional marking phase which writes "marks" to a bitmap; a linking phase which links all references to an object into a single list; and a moving phase which moves all heap objects down in memory deleting holes and updating references to objects.

Again, the stable pointer table must be treated as a "root" during garbage collection.

After the marking phase, we scan through the `_MallocPtr` list releasing those which have not been marked. (This must be done before the linking phase so that the stable pointer table will contain sensible values and can be updated by any calls to `freeStablePtr`.) Each `_MallocPtr` is added to the `_MallocPtr` list as it is moved.

4.3.3 Dual Mode Collection

The dual-mode collector switches between copying (2-space) collection and compacting (1-space) collection according to residency. It is simply a combination of the above.

4.3.4 (Appel) Generational Collection

The Appel generational collector maintains two separate generations: an old generation which is collected by a compacting collector and a new generation which is collected by a copying collector. (When the new generation is collected, it is copied onto the end of the old generation.)

This is essentially just a straightforward combination of the copying and the compacting collectors. However, since the generations are collected separately, we maintain a separate `_MallocPtr` list for each generation. Collecting the new generation transfers all live objects to the old generation and so the new `_MallocPtr` list is emptied and the old list extended.

Care must be taken to ensure that `StgPerformGarbageCollection` performs a full garbage collection: merely flushing the new generation might not release all unreachable objects.

5 Further Work

5.1 Supporting Several Kinds of Malloc Pointer

So far most of our applications have only used malloc pointers as pointers to a single kind of object (eg strings or images). In this situation, defining the function `FreeMallocPtr` is straightforward. As malloc pointers are used more heavily, we anticipate problems with different kinds of object requiring different deallocation routines. An obvious solution is that instead of returning a pointer to an object, a C function should return a pointer to a pair containing a pointer to an object and a pointer to a freeing routine appropriate to that kind of object. (An early implementation stored both the pointer to the object and a pointer to the freeing routine in the Haskell heap. The pointer to the freeing routine was removed after problems persuading C compilers to return pairs reliably.)

5.2 Eliminating a Space Leak

Though individually safe, the provision of both stable pointers and malloc pointers introduces a potential space leak. The reason for this is that it is possible to setup a cyclic structure involving an object in the C world which contains a stable pointer in the Haskell world which contains a malloc pointer to the C object.

This space leak could be eliminated by changing the interface to malloc pointers to allow the C world to take a more active role in garbage collection.

- At start of GC, tell the C world that GC is about to start. (This would allow the C world to clear mark bits (say) on heap objects.)

- During GC, tell the C world whenever a live malloc pointer is found. (This may cause the C world to inform the Haskell world that a stable pointer is live.)
- At end of GC, tell the C world that GC is ending. (This would allow the C world to delete any malloc pointers that have not been marked as live.) (In a generational collector, the C world should be told that all malloc pointers in uncollected (old) generations are live at the end of GC.)

We have not implemented this alternative since the greater complexity (and harder testing) did not seem to be justified by the risk.

5.3 Better Generational Collection

Garbage collecting an entire heap may take some time – it would be nice to be able to perform only as much work as is required to free enough MallocPtr. In a generational garbage collector, this could be done by providing an additional parameter indicating how many generations are to be collected. One could then write:

```
generation = 0;
while(freeSpace < requiredSpace && generation != numGenerations){
  StgPerformGarbageCollection( generation );
  generation += 1;
}
```

Each call to the collector copies the current generation into the next level. Thus, the only overhead of repeatedly calling the collector are the tests that the low-numbered generations are indeed empty.

6 Related Work

Our "malloc pointers" mechanisms bear some resemblance to the "weak references" and "weak arrays" of (the DEC SRC implementation of) Modula 3 [4] and the Objectworks\Smalltalk implementation [13] respectively. Both provide a method of associating a "cleanup procedure" to an object which is called when the associated object "dies".

An essential difference is that the cleanup procedure is written in Modula 3 (respectively Smalltalk). This is possible because both languages support multitasking and associated mechanisms such as semaphores to use it safely. Since the cleanup procedures will, by their nature, involve side-effects, it is not clear that this solution would be appropriate in Haskell.

We note that there is no need to build stable pointers into an imperative language implementation: they are readily implemented by using a global table or list of stable pointers. This provides the same behaviour since the garbage collector will automatically trace all global variables.

7 Summary

There is a wealth of high quality library code freely available for use by imperative programmers. If functional programmers are unable to (or choose not to) use this resource, any claims of greater productivity or higher levels of reuse become nonsense.

Glasgow Haskell's existing foreign language interface allowed one to call simple library functions but was insufficient for creating a reliable interface to a large library featuring involving large lazy polymorphic objects which persist from one function call to the next.

We have described two new types to overcome these problems: `_MallocPtr`s to allow `Haskell` to refer to `C` objects; and `_StablePtr`s to allow `C` to refer to `Haskell` objects.

Acknowledgements

As well as benefiting from the existence of a system on which to implement the ideas in this paper, I have received considerable help from those working on the Glasgow Haskell compiler. In particular, Simon Peyton Jones, Will Partain and Jim Mattson have provided considerable assistance with the compiler modifications described here.

Sigbjorn Finne, David Fraser, Satnam Singh, Paul Smith and Kevin Hammond (all at Glasgow University) provided early versions of Haskell interfaces to a variety of libraries which motivated the development of the ideas in this paper. Ian Poole (at the Medical Research Council Human Genetics Unit, Edinburgh) image-processing requirements prompted the implementation of "Malloc Pointers" described in this paper. Jim Mattson's implementation of interrupt handlers provided yet another motivation for Stable Pointers.

ToDo: Simon: where did the money you paid me come from?

References

[1] AW Appel, Simple generational garbage collection and fast allocation, Software — Practice and Experience 19, 171–183, Feb. 1989.

[2] JF Bartlett, Compacting garbage collection with ambiguous roots, Lisp Pointers 1, 6, 3–12, Apr. 1988.

ToDo: Check that this reference describes weak pointers

[3] HJ Boehm, Space efficient conservative garbage collection, in Proc. ACM Comference on Programming Language Design and Implementation, Albuquerque, 197–206, June 1993.

[4] L Cardelli, J Donahue, L Glassman, M Jordan, B Kalsow, G Nelson, Modula-3 Language Definition, ACM SIGPLAN Notices, 27, 8, 15–42, Aug. 1992.

[5] M Carlsson and T Hallgren, Fudgets: a graphical user interface in a lazy functional language, in Proc. Conference on Functional Programming and Computer Architecture, 1993.

[6] CJ Cheney, A nonrecursive list compacting algorithm, Communications of the ACM 13, 677–678, Nov. 1970.

[7] CLX Common Lisp X Interface, Texas Intruments Incorporated, 1989.

[8] D Fraser, Haskell Defender: implementing arcade games in lazy functional languages, Senior Honours Project, Computing Science Department, University of Glasgow, 1994.

[9] ER Gansner, JH Reppy, eXene, Oct. 1991.

[10] R Harper and P Lee, Advanced languages for systems software: the Fox project in 1994, CMU-CS-94-104, Department of Computing Science, Carnegie Mellon University, Jan 1994.

[11] HBM Jonkers, A fast garbage compaction algorithm, Information Processing Letters 9, 26–30, July 1979.

[12] JT O'Donnell, Data parallel implementation of Extensible Sparse Functional Arrays, Parallel Architectures and Languages Europe, LNCS 694, Springer-Verlag, 68–79, 1993.

[13] ParcPlace Systems, Objectworks\Smalltalk User's Guide (Release 4), 1550 Plymouth Street, Mountain View, California 94043. 1990.

[14] SL Peyton Jones and J Launchbury, Lazy imperative programming, ACM SIGPLAN Workshop in State in Programming Languages, Copenhagen, June 1993.

[15] SL Peyton Jones and J Launchbury, Unboxed values as first class citizens in a non-strict functional language, in Proc. 1991 Conference on Functional Programming and Computer Architecture, Cambridge, Sept. 1991.

[16] SL Peyton Jones and J Launchbury, Lazy functional state threads, in Proc. ACM Comference on Programming Language Design and Implementation, Orlando, June 1994.

[17] SL Peyton Jones and PL Wadler, Imperative functional programming, in Proc. 20th ACM Symposium on Principles of Programming Languages, Charlotte, ACM, Jan 1993.

[18] I Poole and D Charleston, Experience of developing a cervical cytology scanning system using Gofer and Haskell, this volume, September 1994.

[19] AD Reid and S Singh, Implementing fudgets with standard widget sets, in Proc. Glasgow Workshop on Functional Programming, Workshops in Computing Series, Springer-Verlag, July 1993.

[20] PM Sansom, Combining copying and compacting garbage collection, in Proc. Glasgow Workshop on Functional Programming, Workshops in Computing Series, Springer-Verlag, Aug 1991.

[21] PM Sansom and SL Peyton Jones, Generational garbage collection for Haskell, in Proceedings of the 1993 Conference on Functional Programming and Computer Architecture.