Extending Camelot with Mutable State and Concurrency

Stephen Gilmore

Laboratory for Foundations of Computer Science, The University of Edinburgh
King's Buildings, Edinburgh, EH9 3JZ, Scotland

Abstract. Camelot is a resource-bounded functional programming language which compiles to Java byte code to run on the Java Virtual Machine. We extend Camelot to include language support for Camelot-level threads which are compiled to native Java threads. We extend the existing Camelot resource-bounded type system to provide safety guarantees about the heap usage of Camelot threads.

1 Introduction

Functional programming languages allow programmers to express algorithms concisely using high-level language constructs operating over structured data, secured by strong type-systems. Together these properties support the production of high-quality software for complex application problems. Functional programs in strongly-typed languages typically have relatively few programming errors, when compared to similar applications implemented in languages without these beneficial features.

These desirable language properties mean that developers shed the burdens of explicit memory management but this has the associated cost that they typically lose all control over the allocation and deallocation of memory. The Camelot language provides an intermediate way between completely automatic memory management and unassisted allocation and deallocation in that it provides type-safe storage management by re-binding of addresses. The address of a datum is obtained in pattern matching and used in an expression (to store a different data value at that address), overwriting the currently-held value. An affine linear type system prevents addresses from being used more than once in an expression.

The Camelot compiler targets the Java Virtual Machine but the JVM does not provide an instruction to free memory, consigning this to the garbage collector, a generational collector with three generations and implementations of stop-and-copy and mark-sweep collections. The Camelot run-time disposes of unused addresses by adding them to a *free list* of unused memory. On the next allocation caused by the program the storage is retrieved from the head of the free list instead of being allocated by the JVM **new** instruction. When the free list becomes empty the necessary storage is allocated by **new**.

This storage mechanism works for Camelot, but not for Java, because Camelot uses a uniform representation for types which are generated by the

M. Bubak et al. (Eds.): ICCS 2004, LNCS 3038, pp. 306–313, 2004.

compiler, allowing data types to exchange storage cells. This uniform representation is called the *diamond type* [1,2], implemented by a `Diamond` class in the Camelot run-time. The type system of the Camelot language assigns types to functions which record the number of parameters which they consume, and their types; the type of the result; and the number of diamonds consumed or freed.

One example of a situation where type-safe reuse of addresses can be used is in a *list updating* function. As with the usual non-destructive list processing, this applies a function to each element of a list in turn, building a list of the images of the elements under the function. In contrast to the usual implementation of a function such as map, the destructive version applies the function *in-place* by overwriting the contents of each cons cell with the image of the element under the function as it traverses the list.

The following simple function increments each integer in an integer list. The Camelot concrete syntax is similar to the concrete syntax of Caml. Where addresses are not manipulated, as here, a Camelot function can also be compiled by Caml.

```
let rec incList lst =
    match lst with
        []    ->  []
      | h::t  ->  (h + 1) :: incList t
```

This non-destructive version of list processing allocates as many cons-cells as there are elements in the list. With the destructive implementation the storage in the list is reused by overwriting the stored integers with their successors. Thus this version does not allocate any storage.

```
let rec destIncList lst =
    match lst with
        []        ->  []
      | (h::t)@d  ->  ((h + 1) :: destIncList t)@d
```

In a higher-order version of this function, a *destructive map*, we would have the memory conservation property that if the function parameter does not allocate storage then an application of the destructive map function would not either.

Selective use of in-place update in this way can be used to realise *deforestation*, a program transformation which eliminates unnecessary intermediate data structures which are built as a computation proceeds.

As an example of a function which is *not* typable in Camelot we can consider the following one. This function attempts to create a modified copy of a list, interleaved with the original list. The (deliberate) error in implementing this function is to attempt to store the cons cells at the front of the list and the cons cell in second place at the same location, `d`.

```
let rec incListCopy lst =
    match lst with
      []       -> []
    | (h::t)@d -> let tail = ((h + 1) :: t)@d
                  in (h :: tail)@d     (* Error: d used twice! *)
```

This function is faulted by the Camelot compiler with the following diagnostic error message.

```
File "incListCopy.cmlt", line 4-5, characters 18-80:
! ..................let tail = ((h + 1) :: t)@d
!                    in (h :: tail)@d.............
! Variable d of type <> used non-linearly
```

The `destIncList` function above demonstrates storage re-use in Camelot. As an example of programmed control of storage deallocation consider the destructive sum function shown below. Summing the elements of an integer list—or more generally folding a function across a list—is sometimes the last operation performed on the list, to derive an accumulated result from the individual values in the list. If that is the case then at this point the storage occupied by the list can be reclaimed and it is convenient to do this while we are traversing the list.

```
let rec destSumList lst =
    match lst with
        []        ->  0
      | (h::t)@_  ->  h + destSumList t
```

Matching the location of the object against a wildcard pattern (the _ symbol) indicates that this address is not needed (because it is not bound to a name) and thus it can be freed. The `destSumList` function frees the storage which is occupied by the spine of the list as it traverses the list. In a higher-order version such as *destructive fold* we would have the memory reclamation capability that the function passed in as a parameter could also free the storage occupied by the elements of the list, if these were other storage-occupying objects such as lists or trees.

2 Using Threads in Camelot

Previously the JVM had been used simply as a convenient run-time for the Camelot language but a recent extension to Camelot [3] allows the Java namespace to be accessed from a Camelot application. Thus a Camelot application can now create Java objects and invoke Java methods. Additionally, the Camelot language has now been extended with syntactic support for the definition of classes and objects. Figure 1 shows the implementation of a simple clock in Camelot.

This example shows the Camelot syntax for method invocation (`obj#meth()`), field access (`obj#field`) and mutable field update (`f <- exp`). The application simply shows a small window into which is written the current date and time. The parameter to the built-in `sleep` function for threads is given in milliseconds so the application sleeps for a tenth of a second before refreshing the display in the text area on screen with the current date and time.

In the object-oriented fragment of the language the types of parameters and results typically need to be specified by the programmer whereas in the functional part types are inferred. Polymorphic functions are monomorphised in compilation to Java byte code.

```
(* The ticker class, a thread implementing a clock *)
class ticker = java.lang.Thread
with
    field ta : javax.swing.JTextArea

    field formatter : java.text.SimpleDateFormat

    method setTextField (f : javax.swing.JTextArea) : unit =
        ta <- f

    method setFormatting (s : string) : unit =
        formatter <- new java.text.SimpleDateFormat s

    method run() : unit =
        let _ = sleep 100 in
        let now = new java.util.Date() in
        let s = this#formatter#format now in
        let _ = this#ta#setText s in this#run()
end


(* The main function of the application *)
let main args =
    let frame = new javax.swing.JFrame "Camelot clock" in
    let ta = new javax.swing.JTextArea() in
    let clock = new ticker() in
    let _ = clock#setTextField ta in
    let _ = clock#setFormatting "EEE MMM dd hh:mm:ss yyyy" in
    let _ = clock#start() in
    let f = new javax.swing.JFrame() in
    let _ = f#setContentPane ta in
    let _ = f#setSize 180 60 in
        f#setVisible true
```

Fig. 1. An implementation of a threaded clock in Camelot

3 Management of Threads

In designing a thread management system for Camelot our strongest requirement was to have a system which works harmoniously with the storage management system already in place for Camelot. One aspect of this is that the resource consumption of a single-threaded Camelot program can be computed in line with the reasoning explained in Section 1.

In moving from one to multiple threads the most important question with respect to memory usage is the following. Should the free list of storage which can be reused be a single static instance shared across all threads; or should each thread separately maintain its own local instance of the free list?

In the former case the accessor methods for the free list must be synchronised in order for data structures not to become disordered by concurrent write operations. Synchronisation incurs an overhead of locking and unlocking the parent of the field when entering and leaving a critical region. This imposes a penalty on program run-time.

In the latter case there is no requirement for access to the free list to be synchronised; each thread has its own free list. In this case, though, the free memory on each free list is private, and not shared. This means that there will be times when one thread allocates memory (with a Java **new** instruction) while another thread has unused memory on its local free list. This imposes a penalty on the program memory usage, and this form of thread management would lead to programs typically using more memory overall.

We have chosen the former scheme; we have a single static instance of a free list shared across all threads. Our programs will take longer than their optimum run-time but memory performance will be improved. Crucially, predictability of memory consumption is retained.

There are several possible variants on this second scheme which we considered. They were not right for our purposes but might be right for others. One interesting alternative is a hybrid of the two approaches is where each thread had a bounded (small) local free list and flushes this to the global free list when it becomes full. This would reduce the overhead of calls to access the synchronised global free list, while preventing threads from keeping too many unused memory cells locally. This could be a suitable compromise between the two extremes but the analysis of this approach would inevitably be more complicated than the approach which we adopted (a single static free list).

A second alternative would be to implement weak local free lists. In this construction each thread would have its own private free list implemented using *weak references* which are references that are not strong enough by themselves to keep an object alive if no genuine references to it are retained. Weak references are typically used to implement caches and secondary indexes for data structures. Other high-level garbage-collected languages such as O'Caml implement weak references also. This scheme was not usable by us because the Camelot compiler also targets small JVMs on handheld devices and the J2ME does not provide the necessary class (`java.lang.ref.WeakReference`).

The analysis of memory consumption of Camelot programs is based on the consumption of memory by heap-allocated data structures. The present analysis of Camelot programs is based on a single-threaded architecture. To assist with the development of an analysis method for multi-threaded Camelot programs we require that data structures in a multi-threaded Camelot program are not shared across threads. For example, it is not possible to hold part of a list in one thread and the remainder in another. This requirement means that the space

consumption of a multi-threaded Camelot program is obtained as the sum of per-thread space allocation plus the space requirements of the threads themselves.

At present our type system takes account of heap allocations but does not take account of stack growth. Thus Camelot programs can potentially (and sometimes do in practice) fail at runtime with a `java.lang.StackOverflowError` exception because tail calls of Java methods are not optimised by the JVM.

4 A Simple Thread Model for Camelot

To retain predictability of memory behaviour in Camelot we restrict the multi-threaded programming model significantly from that offered by Java's threads.

Firstly, we disallow use of the `stop` and `suspend` methods from Java's threads API. These are deprecated methods which have been shown to have poor programming properties in any case. Use of the `stop` method allows objects to be exposed in a damaged state, part-way through an update by a thread. Use of `suspend` freezes threads but these do not release the objects which they are holding locks on, thereby often leading to deadlocks. Dispensing with pre-emptive thread interruption means that there is a correspondence between Camelot threads and lightweight threads implemented using first-class continuations, `call/cc` and `throw`, as are usually to be found in multi-threaded functional programming languages [4,5].

Secondly, we require that all threads are run, again for the purposes of supporting predictability of memory usage. In the Java language thread allocation (using **new**) is separated from thread initiation (using the `start` method in the `java.lang.Thread` class) and there is no guarantee that allocated threads will ever be run at all. In multi-threaded Camelot programs we require that all threads are started at the point where they are constructed.

Finally, we have a single constructor for classes in Camelot because our type system does not support overloading. This must be passed initial values for all the fields of the class (because the thread will initiate automatically). All Camelot threads except the main thread of control are daemon threads, which means that the Java Virtual Machine will not keep running if the main thread exits.

This simplified idiom of thread use in Camelot allows us to define *derived forms* for Camelot threads which abbreviate the use of threads in the language. These derived forms can be implemented by *class hoisting*, moving a generated class definition to the top level of the program. This translation is outlined in Figure 2.

5 Threads and (Non-)Termination

The Camelot programming language is supported not only by a strong, expressive type system but also by a program logic which supports reasoning about the time and space usage of programs in the language. However, the logic is a

```
let rec threadname(args) =
    let locals = subexps in threadname(args)
...
let threadInstance =
    new threadname(actuals) in ...
⇝
class threadnameHolder(args) = java.lang.Thread
with
    let rec threadname() =
        let locals = subexps in threadname()
    method run() : unit =
        let _ = this#setDaemon(true)
        in threadname()
end
...
let threadInstance =
    new threadnameHolder(actuals) in
let _ = threadInstance#start() in ...
```

Fig. 2. Derived forms for thread creation and use in Camelot

logic of partial correctness, which is to say that the correctness of the program is guaranteed only under the assumption that the program terminates. It would be possible to convert this logic into a logic of total correctness which would guarantee termination instead of assuming it but proofs in such a logic would be more difficult to produce than proofs in the partial correctness logic.

It might seem nonsensical to have a logic of partial correctness to guarantee execution times of programs ("this program either terminates in 20 seconds or it never does") but even these proofs about execution times have their use. They are used to provide a bound on the running time of a program so that if this time is exceeded the program may be terminated forcibly by the user or the operating system because after this point it seems that the program will not terminate. Such *a priori* information about execution times would be useful for scheduling purposes. In Grid-based computing environments Grid service providers schedule incoming jobs on the basis of estimated execution times supplied by Grid users. These estimates are sometimes significantly wrong, leading the scheduler either to forcibly terminate an over-running job due to an under-estimated execution time or to schedule other jobs poorly on the basis of an over-estimated execution time.

6 Conclusions and Further Work

Our programme of research on the Camelot functional programming language has been investigating resource consumption and providing static guarantees of resource consumption at the time of program compilation. Our thread management system provides a layer of abstraction over Java threads. This could allow us to modify the present implementation to multi-task several Camelot threads onto a single Java thread. The reason to do this would be to circumvent the ungenerous thread limit on some JVMs. This extension remains as future work but our present design strongly supports such an extension.

We have discussed a very simple thread package for Camelot. A more sophisticated one, perhaps based on Thimble [6], would provide a much more powerful programming model.

Acknowledgements. The author is supported by the Mobile Resource Guarantees project (MRG, project IST-2001-33149). The MRG project is funded under the Global Computing pro-active initiative of the Future and Emerging Technologies part of the Information Society Technologies programme of the European Commission's Fifth Framework Programme. The other members of the MRG project provided helpful comments on an earlier presentation of this work. The implementation of Java class support in the Camelot compiler is due to Nicholas Wolverson. Java is a trademark of SUN Microsystems.

References

1. Martin Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4):258–289, 2000.
2. Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *Proc. 30th ACM Symp. on Principles of Programming Languages*, 2003.
3. Nicholas Wolverson. O'Camelot: adding objects to a resource-aware functional language. In *On-site proceedings of the Fourth Symposium on Trends in Functional Programming*, Edinburgh, Scotland, September 2003.
4. Edoardo Biagioni, Ken Cline, Peter Lee, Chris Okasaki, and Chris Stone. Safe-for-space threads in Standard ML. *Higher-Order and Symbolic Computation*, 11(2):209–225, 1998.
5. Peter Lee. Implementing threads in Standard ML. In John Launchbury, Erik Meijer, and Tim Sheard, editors, *Advanced Functional Programming, Second International School, Olympia, WA, USA, August 26-30, 1996, Tutorial Text*, volume 1129 of *Lecture Notes in Computer Science*, pages 115–130. Springer, 1996.
6. Ian Stark. Thimble — Threads for MLj. In *Proceedings of the First Scottish Functional Programming Workshop*, number RM/99/9 in Department of Computing and Electrical Engineering, Heriot-Watt University, Technical Report, pages 337–346, 1999.