Estimating the cost of native method calls for resource-bounded functional programming languages

Stephen Gilmore [1]

Laboratory for Foundations of Computer Science, The University of Edinburgh, Edinburgh EH9 3JZ, Scotland

Olha Shkaravska [2]

Ludwig-Maximilians-Universität München, Lehr- und Forschungseinheit Theoretische Informatik, Oettingenstraße 67, D-80538 München, Bundesrepublik Deutschland

Abstract

We address the problem of applying resource-bounded functional programming languages in practice on object-oriented virtual machines which include calls to native methods coded in low-level languages without garbage collection support. We consider the application of a functional language with a high-level type system which incorporates measures of heap space consumption in types on such an execution platform. We supplement the syntactic type inference procedure of the functional language with a separate analysis which estimates the costs of memory leaks incurred by calls to garbage collection-ignorant functions.

1 Introduction

Camelot is a resource-bounded functional programming language which compiles to structured Java byte code which runs on an unmodified Java virtual machine. Camelot functions recycle memory locations in order to reduce the number of object allocations performed, and consequently, the cost of garbage collection. The object allocation cost of evaluating a Camelot function is computed by a syntactic type inference procedure which inspects the abstract structural expression of the function in Damas-Milner style. A type is assigned to a function which includes both the usual surface typing specifying

[1] Email: stg@inf.ed.ac.uk

[2] Email: shkarav@informatik.uni-muenchen.de

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

the types of the function parameters and result and a cost expression specifying the number of objects allocated in terms of the size of the input data structures.

Camelot also provides a foreign function interface which allows an application developer to invoke Java methods. Java in turn has its Java Native Interface (JNI) which allows Java methods to invoke functions which have no equivalent byte code representation. These are typically C routines which manipulate raw memory, access peripherals, or allocate non-Java objects such as operating system resources. In some cases these might be library functions from a proprietary source available only as compiled images with no corresponding program source code. In this case, even if a space leak is detected in the library code it could be impossible to repair without the source code. One can only file a bug report and wait for the maintainers to fix the problem in the next release.

In practice, Camelot programs need to invoke such native methods in order to perform tasks such as building and displaying user interface components. The type inference algorithm which guarantees resource-consumption bounds for applications written entirely in Camelot is not applicable in this situation because not all of the program text is available for inspection and enquiry; and not all of the program text is in the Camelot language in any case. In this paper we propose a complementary resource-based analysis which can be applied to (impure) Camelot applications which invoke Java methods which call C functions.

The invocation of native methods has an impact on the inference of memory heap-allocated by a Camelot application in that the invoked native methods might themselves allocate memory. This memory is allocated in an address space which is inaccessible to the garbage collector of the JVM. If these native methods do not themselves deallocate all of the memory which they have allocated then this effect will manifest itself as a space leak in our application.

Space leaks are not a very significant concern in short-lived applications but they are a highly significant one in long-running applications such as server-side code which is intended to run for weeks or months without failing. As more and more space is leaked in a long-running application the allocation of new memory becomes unproductive. In the short term this leads to performance degradation and in the long term to the application failing with an out-of-memory error.

Thus our analysis is most applicable to server-side code but on the server side is also the right place to use native methods. Native methods are not online portable code in the sense of Java byte code. A native method compiled for a Unix platform will not run under Windows, and *vice versa*. Portability is a significant concern for code which is to run on the client machine because different clients will have different operating systems and hardware but non-portability is not the same issue on the server side. It is frequently the case that a Java application running on a server will be compiled to native

code entirely through the use of a 'way ahead of time' optimising native code compiler such as BulletTrain, SpecialJ or GCJ. Because the analysis which we describe here is specifically tailored for native methods it is to be applied by the code producer as part of software development and tuning, not by the code consumer as part of the code verification process.

We propose a resource-based analysis which predicts resource consumption as a function of time. The main effect of the analysis will be to quantify the severity of the space leaks from native methods with respect to the application overall. This analysis constitutes a *soft* type system for impure server-side Camelot applications. The analysis does not reject any programs but instead classifies the severity of their space leak flaws. Even an application with significant space leak problems might be usable on a large enough server for sufficiently long that regular maintenance on the server would cause it to be likely to be taken down during the time frame in which the application has not leaked enough space for it to be a genuine concern.

In the presence of source-unavailable C functions which must be treated as opaque ("black boxes") by the analysis, the resource-based analysis will necessarily rest on the use of quantified approximation as made formal in the probabilities and average durations used by the model. This leads to the definition of a stochastic process used to represent the impure Camelot application under study.

Structure of this paper:

In the following section we present an overview of Camelot, an extended strict first-order functional programming language. We detail in particular those features which the Camelot language provides for reusing heap-allocated memory. In Section 3 we give an overview of the heap-usage analysis for pure Camelot programs. We discuss this both at the high-level in terms of bounds on heap-space consumption in the functional language and then at the low-level in terms of certifying the byte code into which our high-level programs are compiled by the Camelot compiler. After this we go on in Section 4 to present a novel analysis which shows how impure Camelot programs which call C routines can be analysed for their space consumption behaviour. We describe the tools used to compute the results of the analysis. Conclusions are presented in Section 5.

2 Camelot

The Camelot compiler targets the Java Virtual Machine and provides language support for reusing memory. 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 compiler, allowing data types to exchange storage cells. This uniform representation is called the *diamond type* [4,5], 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.

2.1 Functional and imperative programming

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.

The constructor "::" used below is pronounced "cons". When used on the left hand side of an arrow in the context of a pattern match it will decompose a non-empty list into its head and its tail. When used on the right hand side of an arrow in the context of an expression it will construct a cons-cell to add a new element onto the front of a list.

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

The above is a non-destructive version of the list processing function which allocates as many cons-cells as there are elements in the list.

In the following version ("destIncList") we have a destructive implementation where the storage in the list is reused by overwriting the stored integers with their successors. Thus this version does not allocate any storage.

The constructor "@" used below is pronounced "stored at". When used on the left hand side of an arrow in the context of a pattern match it will obtain the address which a cons-cell is stored at (similarly to the address-of operator `&` of C but without the attendant problems of lack of type safety

or pointer-related memory faults). When used on the right hand side of an arrow in the context of an expression it will store a cons-cell at a given address (thereby overwriting the allocated object which is stored there).

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

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.

2.2 Linearity

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.

2.3 Object-oriented features

Camelot classes contain methods which can invoke functions which are not associated with any class. At the interface between the object sublanguage and the functional sublanguage of Camelot it is necessary to specify the types of the formal parameters of a function. Thus, in the following (contrived) example it is necessary to specify the type of the parameter of the id function.

```
(* An example Camelot class with a method calling a
   function which is defined outside the class  *)
class callExample =
    method myName() : string = id "callExample"
end

let id (s : string) = s
```

As usual, within the functional sublanguage a type inference procedure removes almost all need for the programmer to supply type information.

2.4 Implementation

The Camelot compiler functions in a very different operational mode from most compilers for functional programming languages because it concentrates on having predictable analysis of space consumption and intentionally will not apply optimisations if they are not known to always guarantee to improve space usage with respect to the type-based analysis. The Camelot compiler is also structured in a way which tends to increase our confidence in its correctness by compiling into a structured dialect of Java bytecode called Grail. Grail is structured bytecode in the sense that it is in A-normal form (functions and primitives are applied to values only). The Camelot compiler is available for free download from http://groups.inf.ed.ac.uk/mrg/camelot/.

3 Analysis

To explicate the context for the analysis of impure Camelot applications we first give an overview of the analysis of pure Camelot applications (which do not call functions written in other languages).

The existing inference procedure for Camelot [5] tracks heap allocations in order to report the quantity of heap memory allocated by a Camelot function call. This cost is expressed as a function of the size of the input arguments. The cost may be negative, if presently-allocated memory is freed by the call and returned to the managed free-list in the Camelot run-time.

Other types of memory use are not recorded. Specifically, stack allocation is not tracked by the analysis. Thus, it is possible for Camelot functions to fail for the reason that they recurse sufficiently deeply that they overflow the Java run-time stack. This is possible in principle—nothing in the heap-space analysis prevents it—and it also happens in practice, requiring the developer to rewrite their code to prevent this fault.

3.1 Analysis of Camelot Functions

The Camelot runtime provides a freelist of unused heap units ("diamonds"). When a "diamond" is needed then

- unless the freelist is empty, it is taken from the freelist;
- a JVM **new** (allocation) is performed otherwise.

Deallocation means just returning the "diamond" to the freelist.

The inference procedure is aimed to solve the following task:
to find, if they exist, linear in the size of arguments and a value, functions $\delta_-(\cdot)$ and $\delta_+(\cdot)$, such that if a freelist contains at least $\delta_-(x_1, \ldots, x_m)$ "diamonds" and $f(x_1, \ldots, x_m)$ terminates with the value v, then during the evaluation no new heap units are allocated, and after the evaluation there will be at least $\delta_+(v)$ "diamonds" in the freelist.

The inference rules are designed for a *notated-type* system. For example, a value of a type `L(A, k)` is a list of elements of a type `A`, such that per each element there are k heap units in a freelist, which play a role of a "credit". A judgment $\Gamma,\ n\ \vdash_\Sigma e : A,\ n'$ means that with a signature Σ in a notated typing context Γ and with n extra "diamonds" available, the term e has a notated type A with n' unused "diamonds" left. For the sake of simplicity, we will omit a subscript $_\Sigma$.

Notations play the role of coefficients for the linear bound functions δ_-, δ_+, mentioned in the task. Consider the meaning of a typing judgment in more detail. Let, for example, for an expression e and its free variable x of a type `L(L(int))` one obtains the following typing: $x : \mathtt{L}(\mathtt{L}(\mathtt{int}, 2), 4), 6\ \vdash e : \mathtt{L}(\mathtt{int}, 3), 1$. This means that if x is evaluated to $[l_1, \ldots, l_m]$, and the evaluation of e terminates with a result l and a freelist has size at least $6 + \Sigma_{i=1,\ldots,m}(4 + 2|l_i|) = 6 + 4m + 2\Sigma_{i=1,\ldots,m}|l_i|$ "diamonds", then after the

evaluation there will be at least $1+3|l|$ "diamonds" in the freelist ($|\cdot|$ denotes the length of a list).

As an example of an annotated inference rule consider the rule for the destructive pattern-matching:

$$\frac{\Gamma,\ n \vdash e_1 : A,\ n' \qquad \Gamma,\ x\_h : A,\ x\_t : L(A,\ k),\ n+1+k \vdash e_2 : A,\ n'}{\begin{array}{l}\Gamma,\ x : L(A,\ k),\ \ n\ \vdash_\Sigma \texttt{match x with}\\ \qquad\qquad\qquad\qquad \texttt{Nil ->}\ e_1\\ \qquad\qquad\qquad\qquad \mid \texttt{(x\_h::x\_t)@\_ ->}\ e_2\\ \qquad\qquad\qquad\qquad\qquad\qquad : A,\ n'\end{array}}$$

The sum $1+k$ in the antecedent exposes that to evaluate e_2 one may demand additionally $1+k$ "diamonds". One extra heap unit is provided by deallocation while destructive pattern-matching. The k heap units are for sure in the freelist, just because it's a necessary "credit" of the matched cell. Compare the rule above with the rule for the nondestructive `match`:

$$\frac{\Gamma,\ n \vdash e_1 : A,\ n' \qquad \Gamma,\ x\_h : A,\ x\_t : L(A,\ k),\ n+k \vdash e_2 : A,\ n'}{\begin{array}{l}\Gamma,\ x : L(A,\ k),\ \ n\ \vdash_\Sigma \texttt{match x with}\\ \qquad\qquad\qquad\qquad \texttt{Nil ->}\ e_1\\ \qquad\qquad\qquad\qquad \mid \texttt{(x\_h::x\_t) ->}\ e_2\\ \qquad\qquad\qquad\qquad\qquad\qquad : A,\ n'\end{array}}$$

Here one can provide only k extra "diamonds" guaranteed by a credit of the matched cell.

The whole typing system is sound w.r.t. the *notated operational semantics*, where a relation $m,\ E,\ h \vdash\ e \rightsquigarrow\ v,\ h',\ m'$ means that the evaluation of e with environment E, and heap h terminates in the presence of a freelist of size m, with a value v, modified heap h', and the freelist has m' "diamonds". Here, as usually, an environment (stack) $E :\ Var \rightharpoonup Val$ is a finite partial map from variables to values, heaps $h,\ h' :\ Loc \rightharpoonup Val$ are finite partial maps from location set to values.

Using the notated type system one may obtain a notated signature for a given Camelot function, like "$\texttt{copy} : \texttt{L}(\texttt{A}, \texttt{1}), \texttt{0} - > \texttt{L}(\texttt{A}, \texttt{0}), \texttt{0}$" for a usual, not in-place, `copy`-function. First one augments a type derivation with variables instead of numbers and builds a notated derivation tree for a given function.

Then one collects and solves numerical linear constraints, arising from the side conditions of the rules. Assuming benign sharing of heap regions occupied by values of variables, *a solution of the system of constraints contains desired coefficients of linear heap bounds for the function - they are the notations in its signature.*

The semantical condition called *benign sharing* means that if at a certain point in the evaluation of a program a heap cell is deallocated by a destructive pattern matching, then this cell must not be accessible from the variables occurring in the continuation. This condition is formulated on the level of the operational semantics of the language. Proving a predicate for a compiled Grail image of a given Camelot program (recall that Grail is the structured dialect of Java bytecode which Camelot is compiled into) we have to approximate this condition statically. Until now this has been done by essentially linear usage of Grail variables. We are considering different possibilities to make the approximation of benign sharing less restrictive. There are a few methods available, see for instance, the paper on *usage aspects* [1] or the work on *layered sharing* [7].

In general, the method does not necessary subsume a "freelist size" interpretation for a number in the notated operational semantics. One views it just as a *number of free units available*. Suppose that one has a garbage collector instead of maintaining a freelist. Then, given a functional program f, we find, if exist, linear bounds $\delta_-(\cdot), \quad \delta_+(\cdot)$, such that if $f(x_1, \ \ldots, \ x_m)$ terminates with the value v, and if there are at least $\delta_-(x_1, \ \ldots, \ x_m)$ free cells available, then starting the evaluation with N heap units occupied, if we run the garbage collector every time, when the number of occupied units reaches $N + \delta_-(x_1, \ \ldots, \ x_m)$, during the evaluation the number of occupied units will never exceed $N + \delta_-(x_1, \ \ldots, \ x_m)$.

3.2 Verification of Heap Bounds

One needs to prove that the same functions δ_- and δ_+ define the consumption and gain for a Grail image of the Camelot program under consideration. We define an assertion for Grail programs

$$[\![U, \ n, \ \Gamma \ \blacktriangleright \ A, \ m]\!],$$

the definition of which basically corresponds to the meaning of the high-level type judgment $\Gamma, \ n \ \vdash e : \ A, \ m$, where U contains the set of free variables of e.

To get automatic, syntactically driven, proofs for such predicates we apply *derived rules* which mirror the high-level notated typing rules. The Grail rules are proven a-priori for any syntactical construct of Grail plus freelist managing functions, `make` and `free`. For instance,

$$\frac{\Gamma(x) = \mathbf{L}(\mathtt{int},\, k)}{\mathit{Diamond.Free}(x) : [\![U,\ 0,\ \Gamma \ \blacktriangleright\ \mathbf{L}(\mathtt{int},\, k),\ 1]\!]}$$

One can immediately see technical limitations of our approach. If one has to certify a Grail method `m1`, invoking another method, say `m2`, then eventually the specification of `m2` should be present in a specification table *in the exact form above*. It means that:

- either `m2` is an image of a Camelot function and the specification of `m2` is derivable,
- or the specification is supplied externally.

In order to estimate the reliability of our approach we propose to apply statistical methods. One can consider the "pure" subprograms of impure Camelot application separately and then

- evaluate (statistically) the possible "harm" from native invocations;
- and evaluate the probability of native calls.

In this paper we concentrate on the former task.

4 Analysing impure Camelot applications

Recall that Camelot programs compile to Java bytecode and can execute on an unmodified JVM. The compiled representation of a Camelot program has access to the APIs of the Java platform, and the Java language, in its turn, has a foreign function interface for invoking native methods written in C (the JNI).

We give an example of such interaction, which leads to fatal, from the point of view of heap consumption, consequences.

Consider a Camelot program with a Java method invocation of a method called allocate applicable to object instances of the class outOfMemory. The program below introduces such an object instance (with the name c) and invokes the allocate method in a recursive function called allocLoop.

```
(* A Camelot program which repeatedly calls a Java method *)
val main: string array -> unit
val allocLoop: unit -> unit

let rec allocLoop() =
    let c = new outOfMemory() in          (* ... a Java class *)
    let _ = c#allocate() in               (* ... native method call *)
        allocLoop()      (* ... tail call of this function *)

let main s = allocLoop()
```

The implementation of the Java class `outOfMemory` is given below.

```
/* The Java class used by the Camelot program */
class outOfMemory {
   /* The allocate method is a native method */
   public native void allocate ();

   /* A static block of code executed at class load time */
   static {
      /* Load the shared object library with the compiled
         C implementation of the allocate () method */
      System.loadLibrary("outofmemory")
   }
}
```

The C function `allocate` allocates some memory every time that it is called. Clearly, this process cannot continue forever.

The following C program contains the necessary JNI imports to be invocable from the Java class presented above.

```
/* The C function compiled to a shared library to be used
   by the Java class */
#include <jni.h>
#include "outOfMemory.h"
#include <stdio.h>

JNIEXPORT void JNICALL
Java_outOfMemory_allocate(JNIEnv *env, jobject obj)
{
     printf ("Allocating memory ...");

     /* malloc may fail by returning a null pointer */
     if ((void*)malloc(102400) == NULL) {
          printf (" failed !\n");
     } else {
          printf ("succeeded\n");
     }
     return;
}
```

On a typical Linux platform, this program will allocate memory very rapidly, filling up the available real memory. Then the *Kernel Swap Daemon* (`kswapd`) will be invoked to swap pages of memory out to the swap file. Fairly soon, the swap file fills up and the program may be killed by the operating system (thereby freeing up all of the memory which it claimed) or may simply hang, requiring the program to be killed by the user or the machine to be rebooted.

The effect of attempting to allocate memory when there is no more left to be allocated depends ultimately on the definition of the `malloc` function. The `malloc` function is defined in the C language specification for the standard library to return a null pointer when it cannot allocate the required memory. Potentially any call to `malloc` in a C program must be prepared to deal with a null pointer being returned as a result.

This is form of Camelot application which we seek to analyse via stochastic modelling, (Camelot calling Java, calling C) because it is not amenable to the precise space inference of Hofmann and Jost [5].

4.1 Details of the analysis

Our statistical analysis of impure Camelot applications proceeds by first walking over the program to obtain its control flow graph. During this pass much of the information which relates to the logic of the application can be abstracted away. Arithmetic expression evaluation and other parts of the computation which do not incur heap-allocation events, or call native methods or call pure Camelot functions can be removed in order to formulate a significantly simpler program on which the remainder of the analysis is based.

At this point much of the data dependencies in the program are removed and replaced by statistical approximations to the control flow behaviour. Thus a deterministic conditional expression can be replaced by a probabilistic choice between alternatives and the conditional expression which is contained in the test in the conditional can itself be removed, if its evaluation allocates no storage on the heap either directly or indirectly and calls no native methods. Multi-way branching caused by pattern match evaluation can be treated similarly to give a compound probabilistic choice.

Where a native function is called in the Camelot program we insert a possible failure point representing a fatal out-of-memory error which abruptly terminates the execution of the program. The relative frequency of these also needs to be estimated. Some of these potential failure points can be eliminated if we can determine that the native code routines do not have space leaks using tools such as Dmalloc, Valgrind or others [9].

In order to reflect the strong simplification which has been applied to the program at this point we choose to render it into a different formal language, Ramsey and Pfeffer's *stochastic lambda calculus* [10].

$$\begin{aligned} e ::= {} & x \mid v \mid \lambda x.e \mid e_1\ e_2 \mid \mathbf{let}\ x = e'\ \mathbf{in}\ e \\ \mid {} & \mathbf{choose}\ p\ e_1\ e_2 \\ \mid {} & (e_1, e_2) \mid e.1 \mid e.2 \\ \mid {} & \mathbf{L}\ e_1 \mid \mathbf{R}\ e_2 \mid \mathbf{case}\ e\ e_l\ e_r \end{aligned}$$

The distinctive feature of the stochastic lambda calculus is the probabilistic choice between alternatives (**choose** $p\ e_1\ e_2$ evaluates to e_1 with probability p

and to e_2 with probability $(1-p)$). The remaining constructs are more familiar and include the left- and right-labelling of values in order to distinguish exceptional (or error) results from good values. The **case** construct tests such a label and applies subexpressions e_l and e_r respectively to the carried value.

We make use of the usual datatypes (such as booleans and integers) and built-in functions such as *succ*, *pred*, and *cond*. In addition we use functions such as *sizeof* to return the size of a machine-representation of a type. For readability we write $succ(x)$ as $x+1$ and applications of the three-place conditional function *cond bexp* exp_1 exp_2 as **if** *bexp* **then** exp_1 **else** exp_2. Again for reasons of readability, we will sometimes write **let** $x = e_1$ **in let** $y = e_2$ **in** e_3 as **let** $x = e_1$; **let** $y = e_2$; e_3. Where it is certain that no ambiguity can arise, we will omit the semi-colon in this syntactic form. We will also add redundant parentheses if they seem to be needed to clarify complex compound expressions which enclose multiple sub-expressions.

4.2 Example

Consider a Camelot function of a type `List→unit` which calls a static Java method `fnative` if the length of an input list is even. We suppose that the body of `fnative` contains a native call of a C function with, perhaps, a memory leak.

```
let rec evenOddLength lst =
    match lst with
        []     -> fnative ()
      | h::t   -> match t with
                    [] -> ()
                  | hh :: tt -> (evenOddLength tt)
```

Suppose that we have obtained, say experimentally, the probability p of the length of a list to be even. We can abstract a list to its length and obtain a very simple stochastic λ-term, representing the body of the Camelot function:

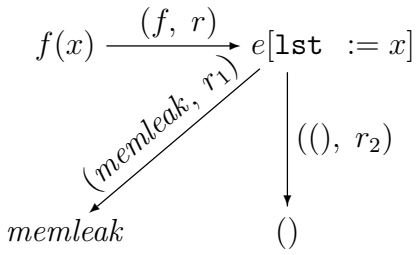
$$choose\ p\ memleak\ ()$$

We have inserted a potential *memleak* point instead of the Java method invocation, assuming that we cannot guarantee that the native call does not contain memory leaks.

We want to map a term in a stochastic λ-calculus onto a Markov chain, the states of which are, in their turn, terms. The transitions of the chain are labeled with pairs $(\alpha,\ r)$, with α for an action and r for a rate. A *rate* is a concept which is more general than a transition probability, in a sense that the sum of two rates for two branching transitions is not necessary equal to 1.

Consider, for instance, a function $f(\texttt{lst})$ in a stochastic λ-calculi which body is presented by the term above: $e \equiv$ *choose* p *memleak* $()$. In this simple case the body of the function does not contain `lst` as a free variable.

The transition graph for the call of this function on a list x will have the form



The rates are proportional to probabilities: $r_i = kp_i$, for some positive k, where $p_1 = p$ and $p_2 = 1 - p$. The rate r is equal to $r_1 + r_2 = k$.

4.3 Second example

Consider a stochastic lambda calculus term which represents a Camelot function which, with probability p_c will invoke a Java method, *javaMethod*, and with probability $1 - p_c$ will invoke another Camelot function, *camelotFn*. We track the changes to the available program memory m, as allocations are made. The outcome is examined in a **case** expression. If all of the memory allocations have been successful then the function will recurse to perform other function or method calls, otherwise the application exits with an out-of-memory error.

$$\begin{aligned}\textbf{let } \mathit{loop} \;&= \lambda m.\ \textbf{let } \mathit{step} \;= \\ &\qquad \textbf{choose } (p_c)(\mathit{javaMethod}\, m)(\mathit{camelotFn}\, m) \\ &\quad \textbf{in case } \mathit{step}\ \mathit{loop}\ \mathit{exit}\end{aligned}$$

Having determined using the Hofmann-Jost type system that the Camelot function does not allocate memory then we could model this using a stochastic lambda calculus term which represents preservation of the current memory level (the current value is labelled with an **L** tag, indicating success).

$$\textbf{let } \mathit{camelotFn} \;= \lambda m.\ \mathbf{L}\ m$$

Examining the Java method we realise that this either invokes a pure Java method (compiled to type-safe, garbage-collected bytecodes), *bytecodeMethod*, or calls a native function (compiled to type-unsafe, memory-unsafe native code), *nativeFn*. With probability p_b it is the bytecode method which is called.

$$\textbf{let } \mathit{javaMethod} \;= \lambda m.\ \textbf{choose } (p_b)(\mathit{bytecodeMethod}\, m)(\mathit{nativeFn}\, m)$$

As with the Camelot function, we may have been able to determine that the bytecode method does not allocate any memory, in which case it too simply preserves the current memory.

$$\textbf{let } \mathit{bytecodeMethod} \;= \lambda m.\ \mathbf{L}\ m$$

Due to its control flow, the native function call will not always lead to a space leak. Sometimes memory will be freed after having been allocated; or on some

calls memory is not allocated. Thus, another probability variable enters the model: the probability with which this application of this function will cause a space leak, p_ℓ.

$$\textbf{let } \mathit{nativeFn} = \lambda m.\ \textbf{choose } (p_\ell)(\mathit{leakSpace}\, m)(\mathit{noSpaceLeak}\, m)$$

The outcome where no space is leaked is similar to the others which we have seen. In the case where space is leaked either a failure outcome or a success outcome is returned, depending on whether or not enough memory is available to satisfy the current allocation request. The denote the type of the currently-requested allocation by alloc_t.

$$\begin{aligned}\textbf{let } \mathit{leakSpace} = \lambda m.\ &\textbf{if } m > \mathit{sizeof}(\mathit{alloc}_t)\\ &\textbf{then L } (m - \mathit{sizeof}(\mathit{alloc}_t))\\ &\textbf{else R } (m)\end{aligned}$$

To make use of this representation of the program it is necessary to seed the stochastic lambda calculus model with estimates of the relative frequency of events in the computation and their duration. These quantities need to be estimated as accurately as is possible at the point of undertaking the analysis of the program. The ideal situation would be to derive these data values from the cost model for the Grail language [2] but our techniques for doing this are not at present sufficiently well advanced and so instead we derive values for these distributions from instrumentation and measurement of our software implementation.

Fortunately, a modern JVM such as SUN's Java HotSpot Virtual Machine release 1.5 has sophisticated intrumentation and measurement capabilities including a redesigned and extended interface to the JVM internals through the Java Virtual Machine Tool Integration API (JVMTI). This generalises and extends the previous JVM Platform Debugger Architecture and JVM Profiling Interface to provide programmatic access to the JVM management and to facilitate instrumentation of Java byte code at run-time.

Having built an abstact representation of the program in a stochastic lambda calculus realisation and having obtained estimates of the frequency of occurrence of events through instrumentation, measurement or other means we are now able to effect our analysis.

4.4 Analysis process

We begin by deriving the reduction sequence which follows from our stochastic lambda calculus term. In any deterministic lambda calculus the branching in such a derivation sequence can be removed by chosing a reduction strategy which identifies a single subterm of the whole which is to be reduced next, and confluence guarantees that the same end result will be obtained. In the stochastic lambda calculus the branching which stems from probabilistic choice cannot be removed and so the reduction path from a given stochastic

lambda calculus term is in general a tree. This tree has (lambda calculus representations of) fatal out-of-memory events at the leaves and we are interested in the duration of the path from the root to one of these leaves.

Now we have formulated the problem as a stochastic process analysis procedure. We wish to compute passage-time quantiles for this activity and so we turn to a stochastic process analysis tool. We have used both the DNAmaca analyser [6] and PRISM [8] for this model.

In Markovian modelling two kinds of analysis deserve mention, because they offer up different insights into the model. The first is a *steady-state analysis*, which looks at the behaviour of the model in the long run, when any initial warm-up period has passed. The second is *transient analysis*, which looks at the behaviour of the model at an instant of time, which might be close to the start of its lifetime and still within reach of any effects of the initial configuration of the system. We perform the latter analysis here.

The results from our analysis tools can be presented either as probability distribution functions (PDFs) or cumulative density functions (CDFs). The meaning of the former is the probability of the measured event happening at this instant in time (so the value of a PDF tends to zero as time tends to infinity because the event becomes more likely to have already happened). The meaning of the latter is the accumulated probability that the measured event has happened plotted against increasing time (so the value of a CDF tends to one as time tends to infinity because the event becomes more likely to have already happened). The graphs presented in Figures 1 and 2 show the results for the above stochastic lambda calculus model presented in Section 4.3.

Since this is a passage-time analysis of a computer application it is legitimate to ask whether the same information could have been obtained simply by repeatedly running the application. The answer is that this information could have been obtained experimentally in this way but that the cost in compute-time would be prohibitively expensive. To plot the above graph from experimental data one would need to re-run the application and accumulate enough observations of the point of failure so that one could meaningfully compute the percentage of times when these failures occurred. Even if we were satisfied with only 100 runs of the application being considered sufficient to allow us to calculate statistically valid results for a long-running program such as the ones considered in the present paper this would mean that the run-time of the series of experiments would be measured in decades of compute time, which is infeasible. The passage-time analysis performed via transient analysis of the underlying continuous-time stochastic process gives us the benefit that we can generalise from short run-times (on a per-function basis) to long run-times (over the passage from system initialisation to system failure).

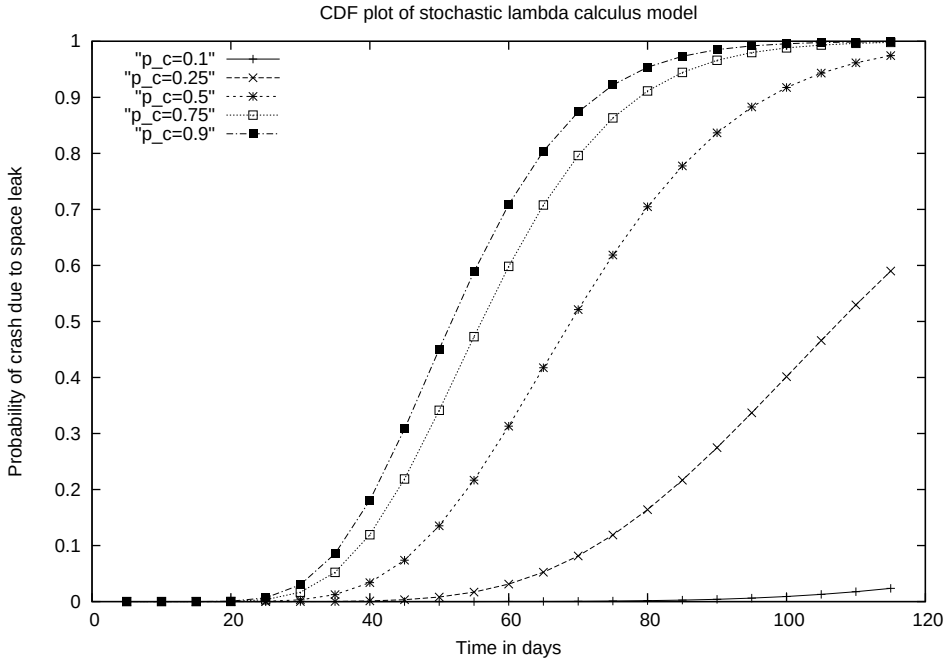


Fig. 1. Cumulative density plot of probability of system fault due to accumulated space leaks varying p_c

4.5 Comparison of results and discussion

Figures 1 and 2 show how the probability of system failure increases as time passes. Thus all of the graphs show the accumulated probability increasing from zero towards one. To produce these plots we held the durations of function call times constant and varied only the probabilities of calling functions of various kinds.

The probability varied in Figure 1 is p_c, the probability of calling a Java method. Space leaks only occur in Camelot applications because Java methods are called so as the probability of these calls being made is reduced, so to is the likelihood of system failure due to memory exhaustion. We hold p_b and p_ℓ constant in this plot (at 0.5).

In Figure 2 we vary p_ℓ, the probability that a called native method leaks space. Variations in this value have a more pronounced effect on the possibility of a crash than might have been expected if we had not taken the trouble to conduct the quantitative analysis which we have done. The conclusion is that reducing the probability of a leak has a marked effect on the longevity of the application, reducing the impact of the leak to the point where it is practicable to tolerate it, in the knowledge that periodic system resets will clear all of the leaked memory in any case. We hold p_b and p_c constant in this plot (at 0.5).

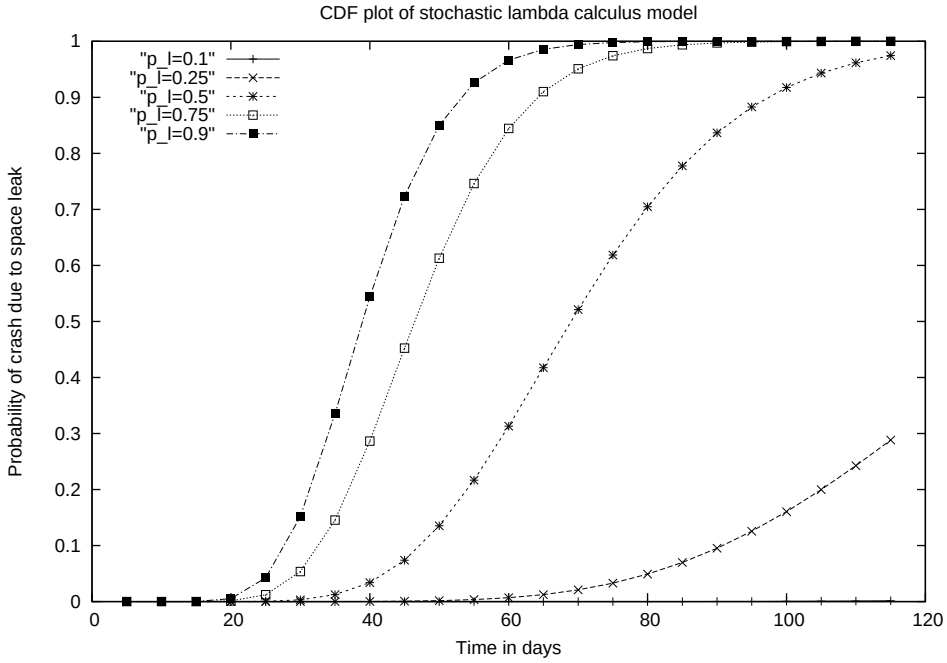


Fig. 2. Cumulative density plot of probability of system fault due to accumulated space leaks varying p_ℓ. The value when $p_\ell = 0.1$ is below 0.0015 even at $t = 115.0$.

5 Conclusions

We have discussed the Camelot programming language, an innovative high-level functional programming language which expresses resource consumption information within its type system. For a principled programming language to be useful, it must also make concessions to practice [3]. The Camelot programming language allows users to invoke Java methods which may in turn invoke native functions in low-level languages such as C. We have taken some time here to discuss how this can interoperate with the high-level language analysis.

The problem discussed is how potential memory leaks, which occur with some probability when a native method is invoked, will impact on the system as a whole. Qualitatively, the end effect is sure: the memory leak will eventually cause system failure. However, we are interested in quantitative analysis here: "How soon will the memory leak cause system failure?" Answers to questions such as these cannot be determined experimentally because the cost of the experimental procedure to determine these (repeatedly re-running a long-running application) is too high. Formal analysis is needed here, and it is useful here.

We have considered the case where some parts of the program (written in the Camelot programming language) are open to inspection and enquiry but others (pre-compiled library images of C applications) are not. Thus the analysis which we undertake must inevitably make use of approximation since the complete behaviour of the C functions is not known. We work with observations of the function behaviour made externally, as can be obtained from the profiling information of time- and space-accurate profilers. These are available for modern JVMs and detail also the cost of native method invocations. Our analysis depends crucially on the accuracy of these approximations. Our use of approximations is in contrast to the function heap-space analysis for the pure subset of the Camelot language due to Hofmann and Jost, which is a precise analysis making no use of approximation.

Seeded with a stochastic process representation of an impure Camelot application obtained via a stochastic lambda calculus intermediate form, the transformation into a continuous-time Markov chain, transformation via uniformisation, transient solution and cumulative density function computation is entirely automatic and can be performed with low computational cost, pointing to the potential usefulness of this method for systems of larger size than those considered here.

The example which we considered was a relatively simple one, with simple control flow. However, the method which we used to analyse this example has greater generality. Mapping into the stochastic lambda calculus allows us express more complex control-flow structures such as partial application, function-passing, dynamic method dispatch, callbacks, event handling, and exceptional completion as found in the functional and object-oriented languages presently in use. This indicates that the method could have practical application in modelling more complex program structures in the impure multi-language application solutions in development today.

Acknowledgements

The authors are 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 feedback on an earlier version of this paper presented at the TFP meeting in Munich was very valuable in helping us to improve this revision. The example of an impure Camelot application given in Section 4 was produced by Matthew Prowse of the MRG project. The implementation of Java class support in the Camelot compiler is due to Nicholas Wolverson. Java is a trademark of SUN Microsystems. The Camelot compiler is available for download from http://groups.inf.ed.ac.uk/mrg/camelot/.

References

[1] David Aspinall and Martin Hofmann. Another type system for in-place update. In *Proc. 11th European Symposium on Programming, Grenoble*, volume 2305 of *Lecture Notes in Computer Science*. Springer, 2002.

[2] Lennart Beringer. Deliverable D1b: cost model, September 2002. Mobile Resource Guarantees deliverable. Available on-line at http://www.lfcs.inf.ed.ac.uk/mrg/internal/deliverables/Costmodel16.pdf.

[3] Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon L. Peyton Jones. Calling hell from heaven and heaven from hell. In *Proceedings of ICFP 1999*, pages 114–125, 1999.

[4] Martin Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4):258–289, 2000.

[5] 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*, pages 185–197. ACM Press, 2003.

[6] W.J. Knottenbelt. Generalised Markovian analysis of timed transition systems. Master's thesis, University of Cape Town, 1996.

[7] Michal Konečný. Functional in-place update with layered datatype sharing. In *TLCA 2003, Valencia, Spain, Proceedings*, pages 195–210. Springer-Verlag, 2003. Lecture Notes in Computer Science 2701.

[8] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM: A hybrid approach. In J.-P. Katoen and P. Stevens, editors, *Proc. 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *LNCS*, pages 52–66. Springer, April 2002.

[9] Jean-Philippe Martin. Memory checkers comparison, 2004. http://www.cs.utexas.edu/users/jpmartin/memCheckers.html.

[10] N. Ramsey and A. Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *Conference Record of the 29th Annual ACM Symposium on Principles of Programming Languages (POPL'02)*, volume 37(1) of *SIGPLAN Notices*, pages 154–165, 2002.