Folding stack memory usage prediction into heap

Brian Campbell*

The static prediction of programs' memory usage has many potential applications in their development and use. In particular, in environments where resources are scarce, such as mobile telephones, a definite upper bound can guarantee that the application will not crash due to a lack of memory.

Hofmann and Jost have presented a system for determining linear bounds on *heap* memory usage for suitable programs in a first order call-by-value functional language. Here we consider obtaining a bound on the *total* memory usage by transforming programs to put information on the heap that would normally be stored on the stack. We also examine the difficulties remaining with this system, and ways in which static prediction could be extended to deal with them.

1 Background

The system Hofmann and Jost presented [3] works on a small first order functional programming language `LF`, using a type system in which the types for data type constructors and functions are annotated by rational numbers which provide an upper bound on the free memory required to execute the program. For example, a function to return the supplied list of integers with each element repeated might be given the type,

```
double :  0, intlist[0|int,#,1] -> intlist[0|int,#,0], 0;
          a          b      c              d      e    f
```

where a gives the number of free cells required by application of the function and f the number released afterwards (both zero in this case), b and c give the number of free cells required per nil and cons constructor present in the argument (one per cons), and d and e the number of free cells released afterwards per nil and cons constructor in the result (none here, the memory was used to construct the result). The annotations can be automatically inferred from constraints in the type system.

The system normally only handles self-contained parts of the program (usually the entire program) because the derived types depend upon how the values are subsequently used. For example, the type given above for the result of the `double` function releases no free memory per element of the list (that is, per cons constructor). This would not be suitable for using the result as an argument to a function requiring free memory proportional to the length of the list, such as another application of `double`. Instead, the inference would provide a higher

*Laboratory for Foundations of Computer Science, School of Informatics, The University of Edinburgh, EH9 3JZ, Scotland. `Brian.Campbell@ed.ac.uk`. This research was supported by the MRG project (IST-2001-33149) which is funded by the EC under the FET proactive initiative on Global Computing.

value for the e annotation, and also for the c annotation in order to balance the books.

To allow memory to be reused pattern matches can be marked as destructive (indicated by `@_` in the example in section 2), which means that the memory the constructor occupies can be freed for reuse once the match has taken place. It is assumed that the placement of such marks will never cause a live value to be deallocated (a condition referred to as *benign sharing*). This can be guaranteed by using a separate analysis to check or generate them [2]. It is also assumed that heap fragmentation will not be a problem (for example, by using a compacting garbage collector, or using fixed size cells chosen to be large enough for any constructor).

When inferring the types the annotations are first given as variables. The typing rules provide constraints that these variables must meet so that the computation will be able to satisfy its memory requirements from the amounts given in the annotations. These constraints form a linear program and solutions to that program provide suitable values for the type annotations.

Should the program have superlinear requirements, or be complex enough that the conservative typing rules cannot capture the linear memory usage, then the generated linear program will be infeasible.

To reduce the complexity of implementing the system a more complete language, Camelot [4], is used for programming, and the analysis performed once the compiler has reduced the language to an LF-like intermediate form. This allows processes such as monomorphisation to take place before the heap use analysis.

2 Transformation

We wish to include the stack space in the bounds inferred using the Hofmann-Jost system. Using a transformation on the program's source code we can make the information stored on the stack into explicit data structures, and then split up the functions so that all function applications become tail calls. The only information not represented by a data structure are the local variables on the top stack frame, whose maximum memory usage could easily be determined by the compiler, or by direct examination of the code. Hence a bound on the total memory usage of both the transformed and original code can be inferred by using Hofmann-Jost on the transformed program. As with the heap-only analysis, the transformation is applied to self-contained parts of the program.

The central idea of the transformation is the introduction of a new data type to hold the stack data. Each constructor will correspond to a function application in the original source code; the choice of constructor is like the return address in a real stack. An extra 'end' constructor is also added to mark the top of the stack so that the end of the top-level function call can be detected.

An extra argument of this type is added to each function, and then for each function application we:

1. add a new constructor to the 'stack' type to contain every (live) local variable including unnamed temporaries and the remainder of the stack;

2. add a corresponding argument, made using the new constructor, to the application;

3. separate out the remaining code in the current function into a new function which takes the contents of the passed-in stack value and the application's result value as its arguments.

When separating out the code in stage 3, sections common to several function calls can be gathered together. Similarly, tail call optimisation can be taken into account by not applying the above steps when there is no remaining code in the calling function.

Finally we introduce a 'stack unwinding' function for each type returned by a function in the original program, which should be called whenever a value of that type is returned without a further function call. This function examines the top value on the stack and calls the appropriate function to continue the computation. The unwind functions can always destructively match the top value because the stack is never shared. Note that all the functions are now mutually recursive.

The process is similar in nature to the transformation to Continuation Passing Style that some compilers use, for example SML/NJ [1]. The 'stack' values (along with the unwind functions) take the place of the continuations' closures avoiding the use of higher order functions. This can be seen as performing the analysis on a lower-level intermediate language than before.

The transformation has been tested by hand on several examples of Camelot code. We consider two below.

Example 1 *A simple function counts all the nodes in a binary tree:*

```
type 'a tree = !Leaf | Node of 'a tree * 'a * 'a tree

let countnodes t = match t with
    Leaf -> 0
  | Node(l,_,r) -> (countnodes l) + 1 + (countnodes r)
```

`!Leaf` *indicates that the* `Leaf` *constructor can be represented without allocating space (by, say, a null reference). The Hofmann-Jost system gives the type*

```
countnodes: 0, tree_1[0|#,int,#,0] -> int, 0;
```

indicating that no free heap memory is required, or known to be released.

The transformation splits up each of the recursive calls, retaining the remainder of each encountered `Node` *in the 'stack'. We add an extra function* `countnodes2` *to provide the same interface as before.*

```
type 'a tree = !Leaf | Node of 'a tree * 'a * 'a tree
type 'a stack = !End | LeftDone of 'a tree * 'a stack
                     | RightDone of int * 'a stack

let countnodes t s = match t with
    Leaf -> unwind_int 0 s
  | Node(l,_,r) -> countnodes l (LeftDone(r,s))
and leftdone v r s = countnodes r (RightDone(v,s))
and rightdone v c s = unwind_int (v + 1 + c) s
and unwind_int v s = match s with
    End -> v
```

```
    | LeftDone(r,s')@_ -> leftdone v r s'
    | RightDone(c,s')@_ -> rightdone v c s'
  let countnodes2 t = countnodes t End
```

Now the overall type is

```
  countnodes2: 0, tree_1[0|#,int,#,1] -> int, 0;
```

requiring one free cell space per node to hold the 'stack'.

A much larger example that has been experimented with is an implementation of the heap sort algorithm. The untransformed program requires (and releases) only a constant amount of extra space when using fixed-size memory allocation, because the memory for the argument list can be reused for the tree and then for the result. With variable-size memory allocation one extra cell is required and released to account for the difference in size between list elements and tree nodes. Perhaps surprisingly, in both cases the transformed version requires only two more cells of memory overall. The Hofmann-Jost analysis of the transformed program treats the stack in the same way as any other heap-allocated structure, so the broken up data structures can be reused to provide memory for the stack frames. Thus the stack space requirements are not reflected in the difference between the inferred bounds on the original and the transformed programs.

3 Problems and further work

When a function requires additional heap space in an untransformed program, that space is normally used for the result or is accounted for in the free space annotation of the result type. In some transformed functions there is no opportunity to do either of these. For instance, the derived type for `countnodes` requires an extra cell of memory per node to keep track of its position in the tree, but only returns an integer. Hence the memory is not used in the result, but the function's type can only show that a constant amount of the free memory is returned afterwards because the annotations cannot express anything more.

Thus later uses of the tree cannot reuse the free memory that was required for `countnodes`. The system has lost track of it after the function call. A similar problem is encountered with accumulating arguments where the memory requirements are also duplicated unnecessarily, and the type system loses track of half of the free space.

The overestimates for stack space can also be more severe in the transformed program. The `countnodes` function only uses logarithmic stack memory when the tree is balanced, but the linear analysis is incapable of determining this and budgets a linear amount instead. Again, the types used in the system are incapable of even expressing the desired effect, but this is potentially harder to solve because it would be necessary to be able to handle conversions between simple types, like lists, and ones with special structure, like balanced trees.

However, the main problem with this analysis is that it only reflects the memory usage when the heap and stack memory are interchangeable, because the results may rely on the memory for dead heap structures being reused for stack space, as in heap sort above. Most systems with a separate stack and heap cannot easily move memory between the two, so in general the inferred requirements are only strict upper bounds on systems where stack data is stored directly on the heap.

An alternative would be to compile the transformed code as this must have the desired behaviour. However, some appealing architectures do not support tail call optimisation which is necessary to keep the remaining 'real' stack space (which is still used for local variables) constant. One such architecture is the Java Virtual Machine, which is the target for the Camelot compiler.

Moreover, the analysis would be of greater use to the programmer if it were more closely related to the source program, because it would allow more direct interpretation of the derived types. For example, all functions in a transformed program have the same return type, regardless of whatever result they actually produce.

To alleviate these difficulties more recent work has been directed to developing a modified version of the type system which analyses stack space requirements alone. The typing rules are changed to reserve space for function applications rather than the use of data type constructors, but the system of annotations to measure those requirements remains intact. The other problems noted above are still present, and are being tackled through alterations to the type system. In addition, the rules need to model tail call optimisation.

The correspondence between the problems in the original system using the transformation and the adapted system for stack alone suggests that solutions for the adapted system could also yield improvements in equivalent situations when considering heap space alone.

4 Summary

The transformation provides a first glimpse of a type system based on Hofmann and Jost's heap space analysis for predicting the total memory usage of first-order functional programs. Using separate, but related, analyses of the heap and stack usage should give a more natural overall system and provide better feedback to the programmer. However, some of the limitations of the current system are more acute when studying stack use, and require further attention.

References

[1] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[2] David Aspinall and Martin Hofmann. Another type system for in-place update. In *Programming Languages and Systems: 11th European Symposium on Programming*, volume 2305 of *Lecture Notes in Computer Science*, pages 36–52. Springer-Verlag, 2002.

[3] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proc. 30th ACM Symp. on Principles of Programming Languages*, New Orleans, 2003.

[4] Kenneth MacKenzie and Nicholas Wolverson. Camelot and Grail: resource-aware functional programming on the JVM. In *Trends in Functional Programing*, volume 4, pages 29–46. Intellect, 2004.