Folding stack memory usage prediction into heap

Brian Campbell

Brian.Campbell@ed.ac.uk
http://homepages.inf.ed.ac.uk/s9746934/

Laboratory for Foundations of Computer Science
The University of Edinburgh[1]

April 2, 2005

[1] 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.

Aim

Hofmann and Jost presented a type-based system for providing linear bounds on the *heap* usage of programs in a first-order call-by-value functional programming language. (POPL'03)

Establishing upper bounds on the memory use of programs is useful for ensuring the reliability of programs, especially in environments where resources are scarce.

- But need to bound the *total* memory use.

A simple transformation resembling a Continuation Passing Style transformation can do this. More importantly, it shows up problems that become more acute for stack space usage.

Roadmap

In this talk:

- A brief overview of Hofmann-Jost system.
- Discuss the transformation used to include stack space in the heap space prediction.
- Use some examples to show problems encountered with stack usage.
- Discuss further work to deal with these problems.

Background

Hofmann and Jost extended the basic type system with:

- positive rational type annotations, representing the amount of free heap memory expected for each constructor in the value;
- side conditions to ensure that providing enough free memory for the annotations is sufficient for execution; and
- the use of explicit 'destruction' hints in the code (which can be checked or inferred separately).

Annotations are inferred using the side conditions as constraints to produce a linear program.

Implemented as part of the Mobile Resource Guarantees project. Analysis is on an intermediate language generated by the compiler. Stages such as monomorphisation to make the analysis simpler.

Example of heap analysis

`double` and `triple` are functions which repeat elements in a list.

If `l` is `[1;2;3]`,

```
double l = [1;1;2;2;3;3],          requiring 3 heap cells;
triple l = [1;1;1;2;2;2;3;3;3], requiring 6 heap cells;
```

Both reuse some of the memory from the supplied list.

Hofmann-Jost can assign the types

```
double : 0, list[int,1] -> list[int,0], 0
triple : 0, list[int,2] -> list[int,0], 0
```

Only the allocated memory is counted—no allowance is made for memory fragmentation. Here we simplify the situation by allocating the same amount of memory for any constructor.

Example of heap analysis II

To type `triple (double l)` the result type of `double` needs to be larger to match `triple`'s argument type:

```
double : 0, list[int,5] -> list[int,2], 0
triple : 0, list[int,2] -> list[int,0], 0
```

The 5 annotation breaks up as

1. 1 for the new element,
2. 2 for the annotation for the new element, and
3. 2 for the annotation for the old element.

So the required type annotations may depend on how the values will be used later on the program—the whole program should be analysed at once.

Transformation

To predict the *total* memory usage for a program, we transform it to make the stack frames explicit data structures.

```
type tree = !Leaf | Node of tree * int * tree
let countnodes t   = match t with
    Leaf ->  0
  | Node(l,_,r) ->    (countnodes l)
                    +  (countnodes r)
                    + 1
```

Transformation

A new data type is introduced to hold the stack data. A constructor holding the live local variables is added for each application.

```
type stack = !End | LeftDone of tree * stack
                  | RightDone of int * stack
type tree = !Leaf | Node of tree * int * tree
let countnodes t s = match t with
    Leaf ->  0
  | Node(l,_,r) ->    (countnodes l (LeftDone(r,s)))
                    +  (countnodes r (RightDone(v,s)))
                    + 1
```

Transformation

The functions are split up so that all function applications are now tail calls. A `unwind` function uses the stack information to call the new functions.

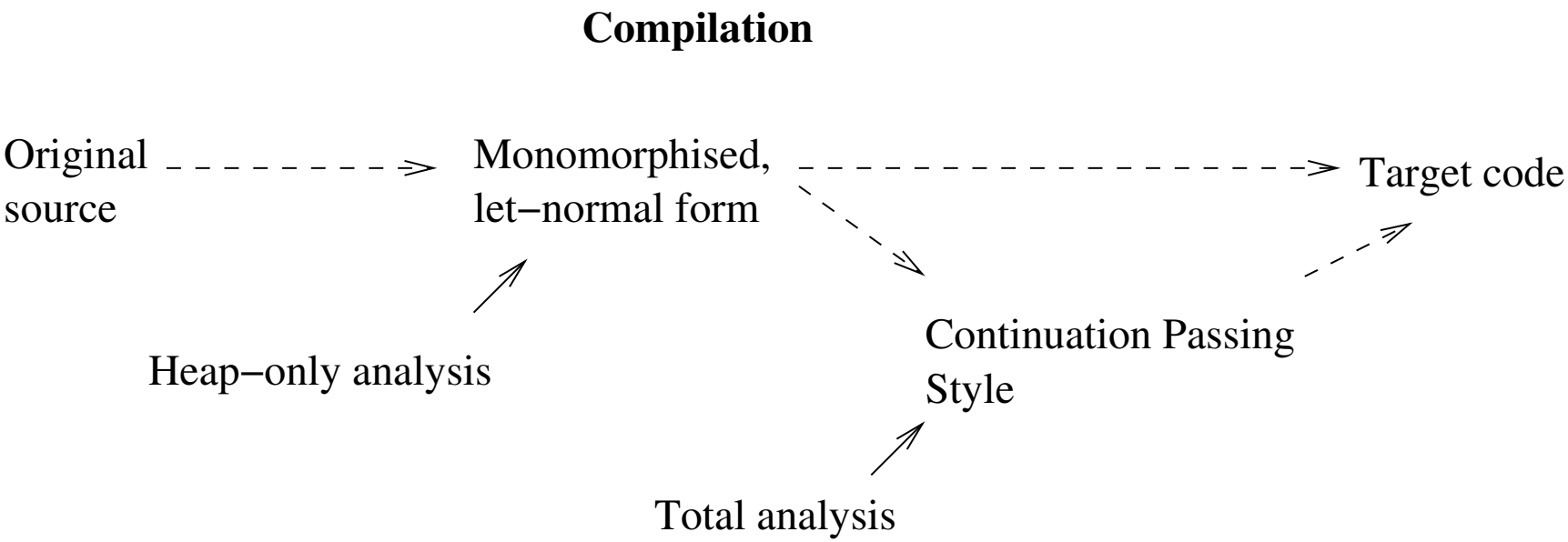
```
type stack = !End | LeftDone of tree * stack
                  | RightDone of int * stack
type tree = !Leaf | Node of tree * int * tree
let countnodes t s = match t with
    Leaf -> unwind 0 s
  | Node(l,_,r) ->    (countnodes l (LeftDone(r,s)))
and leftdone v r s =  (countnodes r (RightDone(v,s)))
and rightdone v c s = unwind (v + 1 + c) s
and unwind v s = match s with
    End -> v
  | LeftDone(r,s')@_ -> leftdone v r s'
  | RightDone(c,s')@_ -> rightdone v c s'
```

The transformation as part of compilation

The transformed code can be used

- to predict the total memory usage of the original program, or
- in place of the original program.

The latter corresponds to producing defunctionalised Continuation Passing Style code as part of the compilation.



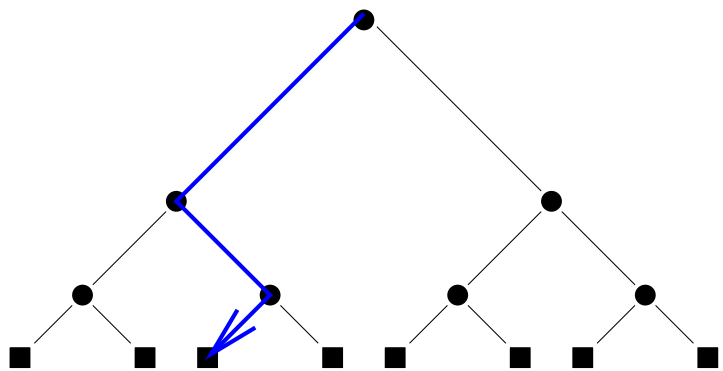
Examples and problems

The tree node counting example shown before has type

```
0, tree[int,1] -> int, 0
```

because one stack frame is required for each level in the tree, and one per node is the lowest annotation that can be given.

This grossly overestimates the requirements; here only the *depth* of the tree is important, not its overall size.



The type is also incapable of showing that the tree's annotation can be used again. With this analysis, the function would have to return the tree to keep the annotation.

Examples and problems II

A functional implementation of heap sort yields the same type

```
1, list[int,0] -> list[int,0], 1
```

for both the heap usage and the total usage! Only a constant overhead is needed because

- Memory for data structures is deallocated during recursion.
- Everything is allocated in fixed-size blocks.

For heap-allocated stack frames this corresponds to the actual memory usage.

For a separate stack and heap, memory may not be interchangable. More useful to consider them as separate resources.

With variable-sized allocation the type annotations increase. But only *logarithmic* space is required.

Further work

Adapt Hofmann-Jost system to measure stack usage directly, dealing with

- 'giving back' annotations
  (type system designed and soundness proof obtained),
- annotations referring to the *depth* of a data structure, rather than the *size*,
- logarithmic annotations so that balanced trees get good bounds,
- (some) tail call optimisation,

and gaining closer correspondence between the analysis and the original source code.

Many of the extensions would also increase the accuracy of corresponding heap space situations.