Prediction of linear memory usage for first-order functional programs

Brian Campbell

Brian.Campbell@ed.ac.uk

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

May 27, 2008

[1] This research was partially supported by the MRG project (IST-2001-33149).

Introduction

Automatically obtained bounds on memory requirements are useful for ensuring reliability and security, and debugging programs.

Hofmann and Jost presented a type-based analysis of linear amounts of *heap* space usage at POPL'03; used for certification in the MRG project.

- Easy to use excessive stack space, even by accident.
- Hence we extend the analysis to *total* memory bounds.
- Want to keep complexity low
- Would like to use a single analysis for both types of memory.

We find that enriching the form of bounds that can be expressed is important for obtaining reasonable bounds.

The Hofmann-Jost heap space analysis

Setting: simple first-order functional programming language with explicit evaluation order and explicit deallocation (use existing safety analysis).

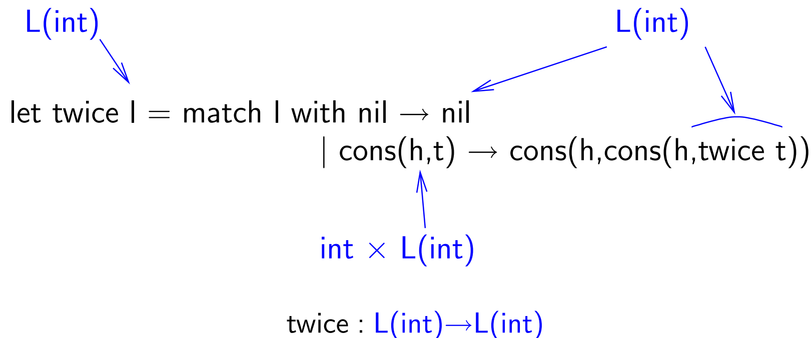
- Add rational annotations to types to represent hypothetical amounts of free memory (*potential*) proportional to the value's size, and to the judgements for constant amounts;
- side conditions in typing rules ensure available potential is sufficient to account for all allocations, and all potential can be traced back to the potential on the input.

Inference is straightforward:

- All the side conditions are linear (in)equalities on rationals;
- use standard linear programming techniques.

Heap example: Repeating elements of a list

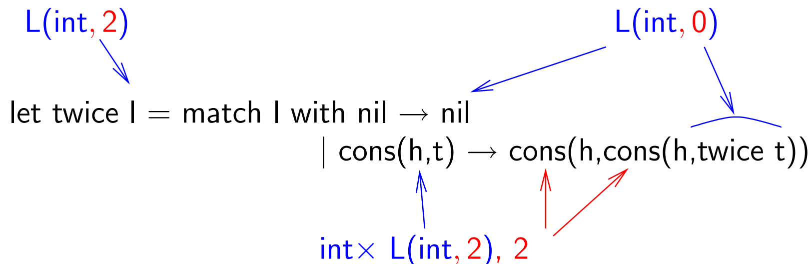
`twice [1,2,3]` $\Rightarrow$ `[1,1,2,2,3,3]`



twice : L(int)→L(int)

Heap example: Repeating elements of a list

```
twice [1,2,3] ⇒ [1,1,2,2,3,3]
```



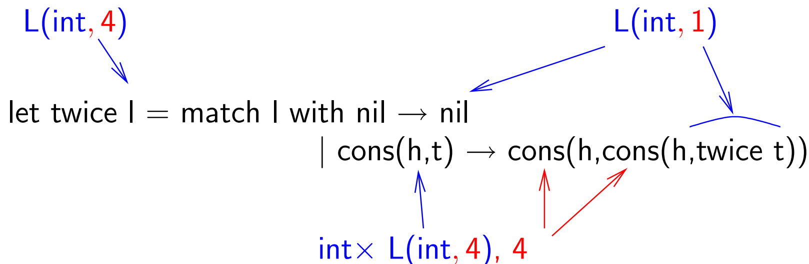
twice : $\mathsf{L}(\mathsf{int}, 2), 0 \rightarrow \mathsf{L}(\mathsf{int}, 0), 0$

So twice x will

- succeed if $2 \times |x| + 0$ cells of memory are free
- ensure $0 \times |\text{twice } x| + 0$ cells are free afterwards

cons is essentially typed as $\mathsf{int} \times \mathsf{L}(\mathsf{int}, 0), 1 \rightarrow \mathsf{L}(\mathsf{int}, 0), 0$

Many possible choices of annotation



twice : $L(\text{int}, 4), 0 \rightarrow L(\text{int}, 1), 0$

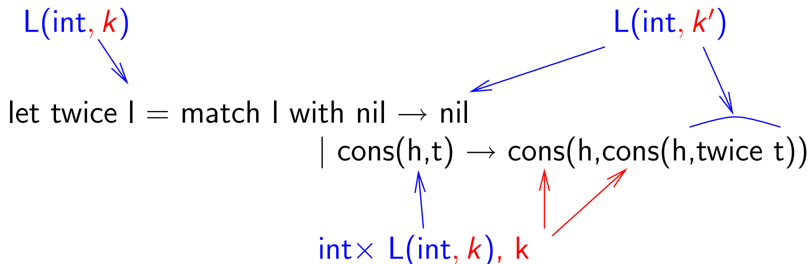
So twice x will

- succeed if $4 \times |x| + 0$ cells of memory are free
- ensure $1 \times |\text{twice } x| + 0 = 2|x|$ cells are free afterwards

cons is essentially typed as $\text{int} \times L(\text{int}, 1), 2 \rightarrow L(\text{int}, 1), 0$

Many possible choices of annotation

```
twice [1,2,3] ⇒ [1,1,2,2,3,3]
```



$$\text{twice} : L(\text{int}, k), 0 \rightarrow L(\text{int}, k'), 0 \quad k \geq 2k' + 2$$

So twice x will

- succeed if $k \times |x| + 0$ cells of memory are free
- ensure $k' \times |\text{twice } x| + 0 = 2k'|x|$ cells are free afterwards

cons is essentially typed as $\text{int} \times L(\text{int}, k'), k' + 1 \rightarrow L(\text{int}, k'), 0$

Heap space rules

The operational semantics include the free memory before and after:

$$\frac{v = (S(h), S(t)) \qquad l \notin \mathrm{dom}(\sigma)}{m + \mathsf{size}(v), S, \sigma \vdash^{g,t} \mathsf{cons}(h, t) \rightsquigarrow l, \sigma[l \mapsto v], m} \;(\textsc{E-Cons})$$

The typing rules have a corresponding constraint on potential:

$$\frac{}{\begin{array}{l}\Gamma, h : T, t : \mathsf{L}(T, k), n \vdash^{g,t}_{\Sigma} \mathsf{cons}(h, t) : \mathsf{L}(T, k), n' \mid \\ \qquad\qquad \{n \geq k + n' + \mathsf{size}(T \otimes \mathsf{L}(T, k))\}\end{array}} \;(\textsc{Cons})$$

The extra k is because the result is one element larger than t.

Adding stack space requirements

- We model stack requirements as fixed frame sizes allocated on function entry;
- and mirror these in the side conditions.

Tail call optimisation is handled by

1. adding tail position flags to judgements,
2. and making the requirements a function on caller, callee and tail position.

Function call rules

$$\frac{f \notin F \qquad \Sigma(f) = (T_1, \ldots, T_p, k \to T, k') \qquad \Phi = \{n \geq k \qquad\qquad , n - k + k' \geq n'\}}{\Gamma, x_1 : T_1, \ldots, x_p : T_p, n \vdash_\Sigma f(x_1, \ldots, x_p) : T, n' \mid \Phi} \; (\textsc{Fun})$$

Heap only — just require enough potential (k) to execute the function.

Function call rules

$$\frac{f \notin F \qquad \Sigma(f) = (T_1, \ldots, T_p, k \rightarrow T, k') \qquad \Phi = \{n \geq k + \mathsf{stack}(f, g, t), n - k + k' \geq n'\}}{\Gamma, x_1 : T_1, \ldots, x_p : T_p, n \vdash_{\Sigma}^{g,t} f(x_1, \ldots, x_p) : T, n' \mid \Phi} \;(\textsc{Fun})$$

- Also require stack space.
- Include caller (g) and tail position flag (t) to account for tail call optimisation.

The problem with temporary usage of stack space

```
let andlists2(l1, l2, l3) =
  let r1 = andlists(l1, l2) in
  let r2 = andlists(l1, l3) in (r1, r2)
```

where `andlists` takes the pointwise and of two lists.

Actual usage: $\min\{|\texttt{l1}|, \max\{|\texttt{l2}|, |\texttt{l3}|\}\}$ frames.

Analysis only provides linear bounds, so expect just $|\texttt{l1}|$.

However. . .

The problem with temporary usage of stack space

```
let andlists2(l1, l2, l3) =
  let r1 = andlists(l1, l2) in
  let r2 = andlists(l1, l3) in (r1, r2)
```

where `andlists` takes the pointwise and of two lists.

Actual usage: $\min\{|\texttt{l1}|, \max\{|\texttt{l2}|, |\texttt{l3}|\}\}$ frames.

Analysis only provides linear bounds, so expect just $|\texttt{l1}|$.

However, the contraction rule adds the requirements with respect to each use of `l1`:

$$\frac{\Gamma, a : T_1, b : T_2, n \vdash^{g,t}_{\Sigma} e : T', n' \mid \Phi \qquad T = T_1 \oplus T_2 \mid \Phi'}{\Gamma, x : T, n \vdash^{g,t}_{\Sigma} e[x/a, x/b] : T', n' \mid \Phi \cup \Phi'} \text{(SHARE)}$$

So we get $2 \times |\texttt{l1}|$.

Give-back annotations

Express post-evaluation potential in terms of the sizes of the result *and* variables in the context.

- Types have extra 'give-back' annotations to express these:

$$\texttt{andlists} : \mathsf{L}(\mathsf{bool}, 1 \rightsquigarrow 1), \mathsf{L}(\mathsf{bool}, 0 \rightsquigarrow 0), 0 \rightarrow \mathsf{L}(\mathsf{bool}, 0 \rightsquigarrow 0), 0$$

- Extra constraints are added to the pattern matching rules to extract the given-back potential.
- A special form of contraction is added to the LET rule which moves potential from the give-back annotation to the normal one.

Give-back annotations on result types

The give-back annotation on the result type indicates an overlap:

$$\texttt{tail} : \mathsf{L}(\mathsf{bool}, 1 \leadsto 1), 0 \to \mathsf{L}(\mathsf{bool}, 1 \leadsto 1), 0$$

- Side condition on new contraction form prevents the use of given-back potential until it is safe.
- Requires that the result is separate from the arguments in question.
- Get separation information from the safety analysis.

Give-back example

```
let andlists2(l1, l2, l3) =
  let r1 = andlists(l1, l2) in
  let r2 = andlists(l1, l3) in (r1, r2)
```

- r1 is freshly allocated
- New let contraction allows all appearences of l1 to be typed

$$L(\text{bool}, 1 \rightsquigarrow 1).$$

- So overall bound |l1| found.

Soundness and inference

- The statement for the induction is slightly delicate due to the ‘overlap’ in the post-evaluation potential;
- key idea is to use per-heap-location information in statement, and the separation information to rule out overlap at the key point.

Inference is similar to before:

1. construct typing, using separation information to decide which form of contraction to use;
2. collect constraints;
3. apply LP solver.

Small linear increase in the number of constraints.

Implemented in ocaml as an extension to Jost’s `lfd_infer` tool.

Further Work

Allow bounds to include maxima and data structure depth.

- Much progress made
- More complex; stack space only so far

Extend the analysis to cope with more language features.

- e.g., merge with Jost's heap space analysis for higher order functions.