Amortised memory analysis using the *depth* of data structures

Brian Campbell

Brian.Campbell@ed.ac.uk

Laboratory for Foundations of Computer Science

March 12, 2008

Outline

Hofmann-Jost Heap Memory Analysis

Example for stack space

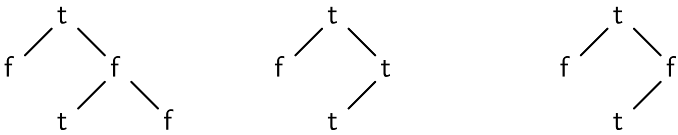
The Stack Analysis Type System

The Stack Space Inference

Further Work

Heap memory example

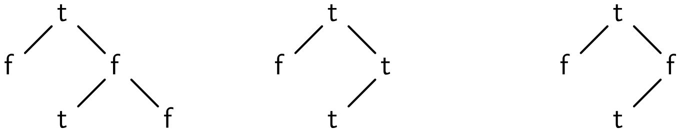
The `andtrees` function computes the pointwise 'and' of two boolean trees (up to the smaller tree):



$\text{bool tree} \quad \times \quad \text{bool tree} \quad \rightarrow \quad \text{bool tree}$

Heap memory example

The `andtrees` function computes the pointwise ‘and’ of two boolean trees (up to the smaller tree):

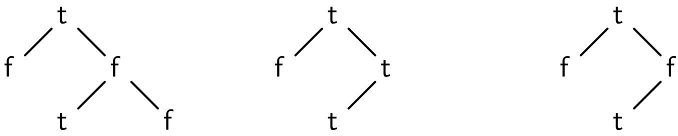


$$\text{bool tree } [1] \quad \times \quad \text{bool tree } [0], \quad 0 \quad \longrightarrow \quad \text{bool tree } [0], \quad 0$$

- means `andtrees t1 t2` uses no more than $|\texttt{t1}|$ units of space.

Heap memory example

The `andtrees` function computes the pointwise ‘and’ of two boolean trees (up to the smaller tree):

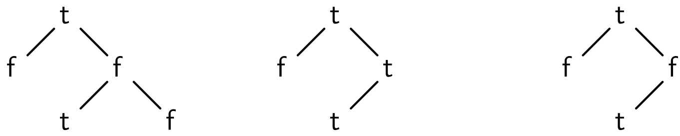


$$\text{bool tree } [1] \quad \times \quad \text{bool tree } [0], \quad 0 \quad \rightarrow \quad \text{bool tree } [0], \quad 0$$
$$\text{bool tree } [0] \quad \times \quad \text{bool tree } [1], \quad 0 \quad \rightarrow \quad \text{bool tree } [0], \quad 0$$

- means `andtrees t1 t2` uses no more than |`t1`| units of space.
- The typings (and bounds) are not unique. |`t2`| is also sufficient.

Heap memory example

The `andtrees` function computes the pointwise 'and' of two boolean trees (up to the smaller tree):



$$\text{bool tree } [2] \quad \times \quad \text{bool tree } [0], \quad 4 \quad \rightarrow \quad \text{bool tree } [1], \quad 4$$

```
let x = andtrees y z in ...
```

- Signatures also 'translate' requirements:
- If ... requires $|x| + 4$ units, then $2 \times |y| + 4$ is sufficent for both allocation and $|x| + 4$ later.

Hofmann-Jost rules

$$\frac{n \geq \text{size(bool tree node)} + k + n'}{l : \text{bool tree}[k], r : \text{bool tree}[k], v : \text{bool}, n \vdash \text{node}(l, v, r) : \text{bool tree}[k], n'} \quad (\textsc{Node})$$

means that if we have

$$|l| \times k + |r| \times k + n$$

units of free memory then we can allocate the node and end up with

$$|\text{node}(l, v, r)| \times k + n' = (1 + |l| + |r|) \times k + n'.$$

Hofmann-Jost inference

$$\frac{n \geq \text{size(bool tree node)} + k + n'}{l : \text{bool tree}[k], r : \text{bool tree}[k], v : \text{bool}, n \vdash \text{node}(l, v, r) : \text{bool tree}[k], n'} \ (\textsc{Node})$$

- Collect constraints from typing rules;
- Solve linear program, minimising the bound.

Hofmann-Jost as an amortized analysis

$$\frac{n \geq \text{size(bool tree node)} + k + n'}{l : \text{bool tree}[k], r : \text{bool tree}[k], v : \text{bool}, n \vdash \text{node}(l, v, r) : \text{bool tree}[k], n'} \text{(Node)}$$

The type annotations define *potential* functions

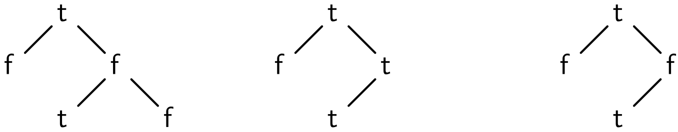
$$\Upsilon_\Gamma(l, r) = |l| \times k + |r| \times k + n$$

for the context, and for the result:

$$\Upsilon_R(R) = |R| \times k + n'.$$

Constraint ensures that the allocation is accounted for by a drop in potential. (See *Physicist's view* in Tarjan 1985)

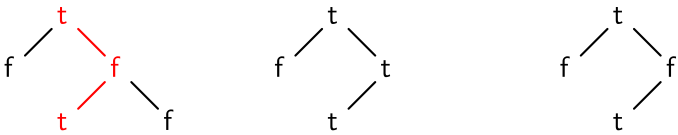
Example with stack space



$$\text{bool tree } [1] \;\times\; \text{bool tree } [0], \quad 0 \;\longrightarrow\; \text{bool tree } [1], \quad 0$$
$$\text{bool tree } [0] \;\times\; \text{bool tree } [1], \quad 0 \;\longrightarrow\; \text{bool tree } [1], \quad 0$$

- means `andtrees t1 t2` uses at most |`t1`| (or |`t2`|) units of stack space.
- Stack space is reusable.

Example with stack space



bool tree [1] $\times$ bool tree [0], 0 $\rightarrow$ bool tree [1], 0
bool tree [0] $\times$ bool tree [1], 0 $\rightarrow$ bool tree [1], 0

- means `andtrees t1 t2` uses at most $|\texttt{t1}|$ (or $|\texttt{t2}|$) units of stack space.
- Stack space is reusable.
- But now we want to use the depth to get a better bound (i.e., $|\texttt{t1}|_{\mathrm{d}}$).

Developing an analysis with maximums

Previously we just added all the 'potential' from the context:

$$\begin{array}{cccccccc} l:\text{bool tree }[k], & & r:\text{bool tree }[k], & & v:\text{bool}, & n \vdash \ldots \\ |l| \times k & + & |r| \times k & + & 0 & +n \end{array}$$

Now we introduce a second context former to denote 'max' (;):

$$\begin{array}{lccccccc} & (l:\text{bool tree }[k]; & & r:\text{bool tree }[k]; & & v:\text{bool}), & n \vdash \ldots \\ \max\{ & |l|_d \times k & , & |r|_d \times k & , & 0 & \}+n \end{array}$$

- Treat tree types as 'folded up' version of above context.
- So t: bool tree $[k]$ has potential of $|t|_{\mathrm{d}} \times k$.
- Note that contexts are now trees.

Inspired by O'Hearn's Bunched Typing.

Bunched Contexts

$$\Gamma := \cdot \mid x\colon T \mid \Gamma, \Gamma \mid \Gamma; \Gamma \mid k$$

$$\begin{aligned}
\Upsilon_\Gamma(S, \cdot) &= 0 \\
\Upsilon_\Gamma(S, x\colon T) &= \Upsilon_\mathsf{T}(S(x), T) \\
\Upsilon_\Gamma(S, (\Gamma_1, \Gamma_2)) &= \Upsilon_\Gamma(S, \Gamma_1) + \Upsilon_\Gamma(S, \Gamma_2) \\
\Upsilon_\Gamma(S, (\Gamma_1; \Gamma_2)) &= \max\{\Upsilon_\Gamma(S, \Gamma_1), \Upsilon_\Gamma(S, \Gamma_2)\} \\
\Upsilon_\Gamma(S, k) &= k
\end{aligned}$$

So $(x\colon T_1, k); y\colon T_2$ has potential

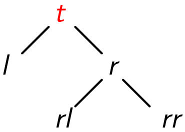
$$\max\{|x|_{\mathrm{d}} + k, |y|_{\mathrm{d}}\}.$$

Unfolding trees in the context

$\Gamma()$ is a context with a 'hole'.

$$\frac{\Gamma(\cdot) \vdash e_1 : T, k' \qquad \Gamma((l : \text{bool tree}[k]; r : \text{bool tree}[k]; v : \text{bool}), k) \vdash e_2 : T, k'}{\Gamma(t : \text{bool tree}[k]) \vdash \text{match } t \text{ with } \begin{array}{l} \text{leaf} \rightarrow e_1 \\ \mid \text{node}(l, v, r) \rightarrow e_2 : T, k' \end{array}} \ (\textsc{TreeMatch})$$

$\rightarrow$



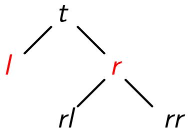
$t : \text{bool tree}[k]$

Unfolding trees in the context

$\Gamma()$ is a context with a 'hole'.

$$\frac{\Gamma(\cdot) \vdash e_1 : T, k' \qquad \Gamma((l : \text{bool tree}[k]; r : \text{bool tree}[k]; v : \text{bool}), k) \vdash e_2 : T, k'}{\Gamma(t : \text{bool tree}[k]) \vdash \text{match } t \text{ with } \begin{array}{l} \text{leaf} \to e_1 \\ \mid \text{node}(l, v, r) \to e_2 : T, k' \end{array}} \quad (\textsc{TreeMatch})$$

$\longrightarrow$



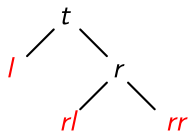
$(l : \text{bool tree}[k]; r : \text{bool tree}[k]; v : \text{bool}), k$

Unfolding trees in the context

$\Gamma()$ is a context with a 'hole'.

$$\frac{\Gamma(\cdot) \vdash e_1 : T, {\color{blue}k'} \qquad \Gamma((l : \text{bool tree}{\color{blue}[k]}; r : \text{bool tree}{\color{blue}[k]}; v : \text{bool}), {\color{blue}k}) \vdash e_2 : T, {\color{blue}k'}}{\Gamma(t : \text{bool tree}{\color{blue}[k]}) \vdash \text{match } t \text{ with } \begin{array}{l} \text{leaf} \rightarrow e_1 \\ \mid \text{node}(l, v, r) \rightarrow e_2 : T, {\color{blue}k'} \end{array}} \; (\textsc{TreeMatch})$$

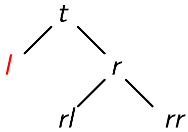
$\rightarrow$



$({\color{red}l} : \text{bool tree}{\color{blue}[k]}; (({\color{red}rl} : \text{bool tree}{\color{blue}[k]}; {\color{red}rr} : \text{bool tree}{\color{blue}[k]}; rv : \text{bool}), {\color{blue}k}); v : \text{bool}), {\color{blue}k}$

Folding trees in the context

$$\frac{k' \geq k + k''}{(l : \text{bool tree}[k]; r : \text{bool tree}[k]; v : \text{bool}), k' \vdash \text{node}(l, v, r) : \text{bool tree}[k], k''} \;(\textsc{Node})$$



$$(l : \text{bool tree}[k]; ((rl : \text{bool tree}[k]; rr : \text{bool tree}[k]; rv : \text{bool}), k); v : \text{bool}), k$$

New rules

We need to be able to manipulate contexts to get the right shape. Hence new rules such as:

$$\frac{\Gamma(\Delta') \vdash e : T, n' \qquad \Delta \cong \Delta'}{\Gamma(\Delta) \vdash e : T, n'} \quad (\equiv)$$

$$\Gamma, (\Delta; \Delta') \cong (\Gamma, \Delta); (\Gamma, \Delta') \quad \text{(distribution)}$$

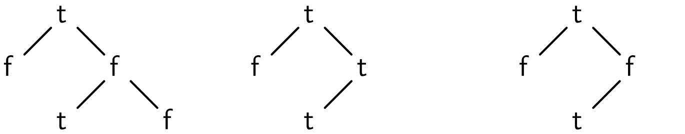
$$\Gamma \cong \Gamma; \Gamma \quad \text{(max-contraction)}$$

$$\Gamma \cong q\Gamma, (1-q)\Gamma \quad q \in [0,1] \quad \text{(plus-contraction)}$$

All preserve the 'potential' (i.e. give the same potential function).

Example with stack space

Function signatures are also 'structured'.



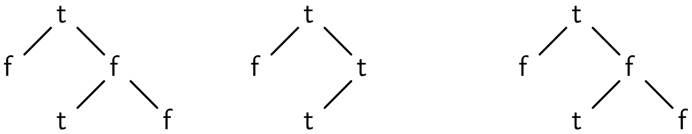
bool tree[1] ; bool tree[0] $\rightarrow$ bool tree[1]

bool tree[0] ; bool tree[1] $\rightarrow$ bool tree[1]

- means `andtrees t1 t2` uses at most $|\texttt{t1}|_d$ or $|\texttt{t2}|_d$ units of stack space.

Example with stack space

Function signatures are also 'structured'.

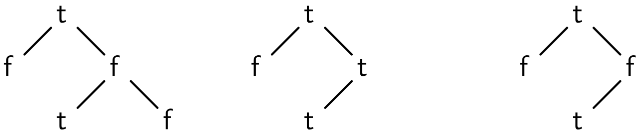


bool tree[1] ; bool tree[1] $\rightarrow$ bool tree[1]

- means `andtreesmax t1 t2` uses at most $\max\{|\texttt{t1}|_d, |\texttt{t2}|_d\}$ units of stack space.
- We can now also type a version of `andtrees` which keeps all the nodes which are only in one of the arguments.

Example with stack space

Function signatures are also ‘structured’.



bool tree[1] ; bool tree[1] $\rightarrow$ bool tree[1]

$$\frac{\Sigma(f) = \Gamma \rightarrow T, k_1 \qquad k \geq \mathrm{stack}(f) \qquad k + k_1 \geq k'}{\Gamma[x_1, \ldots, x_p/\mathrm{namesof}(\Gamma)], k \vdash f(x_1, \ldots, x_p) : T, k'} \ (\textsc{Fun})$$

Extra benefit from maxima

```
let maybetail(l,b) =
  match l with cons(h,t)' ->
    if b then t else l
```

In heap analysis need to sum requirements because of use of contraction at `match`. Doubles the bound unnecessarily.

- In depth type system we can use max-contraction.
- So requirement goes $|l| \Rightarrow \max\{|l|, |l|\} \Rightarrow \max\{|t|, |l|\}$.
- Context goes $l : \text{list} \Rightarrow l : \text{list}; l : \text{list} \Rightarrow t : \text{list}; l : \text{list}$

What about let?

$$\frac{\Gamma_1 \vdash e_1 : T_1 \qquad \Gamma_2, x : T_1 \vdash e_2 : T_2}{\Gamma_1, \Gamma_2 \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : T_2}\ (\textsc{Boring Let})$$

We don't necessary want to sum requirements, and we do want to preserve tree structure.

- Really want to use a small subcontext Γ_1 inside $\Gamma_2(\Gamma_1)$, replace with $x : T_1$.
- This kind of thing appears in Bunched Typing too.
- Prepared to forget about bounding heap space.

Local context replacement

For stack space we always get back the memory that we put in.
So we have the same amount of 'free memory' at e_2 as the start of the let expression.
Thus we can change the form of potential function so long the values it produces cannot increase.

$$\frac{\Gamma_1 \vdash e_1 : T_1, n_1 \qquad \Gamma(x : T_1, n_1) \vdash e_2 : T, n'}{\Gamma(\Gamma_1) \vdash \text{let } x = e_1 \text{ in } e_2 : T, n'} \quad (\textsc{Let})$$

- Relies on stack discipline.
- Heap allocation is harder:
  If we have $\Gamma_1; \Gamma_2$ and allocate in e_1, then we might not have enough left for Γ_2.

Local replacement example

$$\text{let } x = \text{node}(l, r, v) \text{ in } e_2$$

Say e_2 requires $\max\{|x|_\mathrm{d} + |y|_\mathrm{d}, 5\}$.

$$(x\!:\!\text{bool tree}[1]; y\!:\!\text{bool tree}[1]), 5 \vdash e_2 : T, k$$

The typing of e_1 can 'translate' the x part of the bound:

$$(l\!:\!\text{bool tree}[1]; r\!:\!\text{bool tree}[1]; v\!:\!\text{bool}), 1 \vdash \text{node}(l, r, v)\!:\!\text{bool tree}[1], 0$$

So replace x in e_2's context:

$$((l\!:\!\text{bool tree}[1]; r\!:\!\text{bool tree}[1]; v\!:\!\text{bool}), 1, y\!:\!\text{bool tree}[1]), 5 \vdash e_2 : T, k$$

$$\max\{\max\{|l|_\mathrm{d}, |r|_\mathrm{d}\} + 1 + |y|_\mathrm{d}, 5\}$$

But for **heap** part of the 5 units may be used up in the allocation.

Stack space inference

- Would like to take advantage of linear programming again
- But new context manipulation rules are not syntax-directed

We add an extra stage to the inference process:

source program (plain types)
$\downarrow$
additional terms for context manipulation
$\downarrow$
bound (using linear programming)

Assume context structure given for function signatures to make problem more tractable.

Basic ideas for inference

- Work from the leaves of the expression outwards.
- At every stage, keep track of a generated context derived from subexpressions and the typing rule.

$$\frac{\Gamma \vdash e_1 \mapsto \Gamma_1 \qquad \Gamma \vdash e_2 \mapsto \Gamma_2}{\Gamma \vdash \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mapsto \Gamma_1; \Gamma_2; x : \mathsf{bool}}$$

Need to add context manipulation at two points:

1. where binding occurs, to deal with contraction, etc;
2. to make the generated context match the function signature.

Expanding contexts

We can simplify the problems by distributing over ';' as far as possible to get a maximum-of-sums context:

$$
\begin{aligned}
((((a, b); c), d); e), f &\cong (((a, b); c), d, f); (e, f) \\
&\cong (a, b, d, f); (c, d, f); (e, f)
\end{aligned}
$$

$$\max\{\max\{a + b, c\} + d, e\} + f = \max\{a + b + d + f, c + d + f, e + f\}$$

(Potentially exponential, but contexts are small. May be possible to reduce amount of expansion.)

Binding

We can pick out the plus-bunches of the expanded context involving the bound variable and factor them out:

$$\Gamma_1; \ldots; \Gamma_n; (x : T, \Gamma_{n+1}); \ldots; (x : T, \Gamma_m) \vdash e : T, n'$$
$$\downarrow$$
$$\Gamma_1; \ldots; \Gamma_n; (x : T, (\Gamma_{n+1}; \ldots; \Gamma_m)) \vdash \text{distribute}(n+1, e) : T, n'$$

- Add contraction rules as necessary.
- The multivariable case is similar, but may require more approximation.

Generated contexts vs function signatures

- Expand both ends;
- pick out bunch(es) in the expanded signature containing all the variables of each generated bunch;
- weaken away any extras;

duplicating the signature's bunches as necessary.

Generated context $(a; b), (c; d)$

function signature $a, b, ((c, e); (d, e))$

Generated contexts vs function signatures

- Expand both ends;
- pick out bunch(es) in the expanded signature containing all the variables of each generated bunch;
- weaken away any extras;

duplicating the signature's bunches as necessary.

$$
\begin{array}{rcl}
\text{Generated context} & & (a; b), (\color{red}{c}; d) \\
\text{(expanded)} & \cong & (a, \color{red}{c}); (a, d); (b, \color{red}{c}); (b, d)
\end{array}
$$

$$
\begin{array}{rcl}
\text{(expanded)} & & (a, b, \color{red}{c}, e); (a, b, d, e) \\
\text{function signature} & \cong & a, b, ((\color{red}{c}, e); (d, e))
\end{array}
$$

Generated contexts vs function signatures

- Expand both ends;
- pick out bunch(es) in the expanded signature containing all the variables of each generated bunch;
- weaken away any extras;

duplicating the signature's bunches as necessary.

$$
\begin{array}{rcl}
\text{Generated context} & & (a; b), (c; d) \\
\text{(expanded)} & \cong & (a, c); (a, d); (b, c); (b, d) \\
\\
\text{(max-contract)} & & (a, b, c, e); (a, b, c, e); (a, b, d, e); (a, b, d, e) \\
\text{(expanded)} & \cong & (a, b, c, e); (a, b, d, e) \\
\text{function signature} & \cong & a, b, ((c, e); (d, e))
\end{array}
$$

Generated contexts vs function signatures

- Expand both ends;
- pick out bunch(es) in the expanded signature containing all the variables of each generated bunch;
- weaken away any extras;

duplicating the signature's bunches as necessary.

$$\begin{array}{rcl}
\text{Generated context} & & (a; b), (c; d) \\
\text{(expanded)} & \cong & (a, c); (a, d); (b, c); (b, d) \\
\text{(rearrange)} & \cong & (a, b, c, e); (a, b, d, e); (a, b, c, e); (a, b, d, e) \\
\text{(max-contract)} & \cong & (a, b, c, e); (a, b, c, e); (a, b, d, e); (a, b, d, e) \\
\text{(expanded)} & \cong & (a, b, c, e); (a, b, d, e) \\
\text{function signature} & \cong & a, b, ((c, e); (d, e))
\end{array}$$

Weakening bridges the gap between lines 2 and 3.

Generated contexts vs function signatures

- Expand both ends;
- pick out bunch(es) in the expanded signature containing all the variables of each generated bunch;
- weaken away any extras;

duplicating the signature's bunches as necessary.

Nasty case:

Generated	a, b
Signature	$a; b$

Use rule corresponding to $\max\{x, y\} \geq \frac{1}{2}(x + y)$.
Sound, but sometimes imprecise.

Fixed amounts of potential in contexts

Fixed amounts may appear anywhere in the context, but are not explicitly introduced by binding.

- When expanding contexts to the maximum-of-plus form, we can ensure that every plus-bunch has exactly one fixed amount

$$(x_{11} : T_{11}, \ldots, k_1); \ldots; (x_{n1} : T_{n1}, \ldots, k_n)$$

- When partitioning the expanded context for a binding, add context manipulation terms so that the resulting fixed amounts can come from the binding *or* the subcontexts:

$$\Gamma_1; \ldots; \Gamma_n; (x : T, \Gamma_{n+1}, k_{n+1}); \ldots; (x : T, \Gamma_m, k_m)$$
$$\downarrow$$
$$\Gamma_1; \ldots; \Gamma_n; \left(x : T, k, ((\Gamma_{n+1}, k'_{n+1}); \ldots; (\Gamma_m, k'_m))\right)$$

The typing rules require that $k + k'_i \geq k_i$.

Finishing inference

source program (plain types)

$\downarrow$

additional terms for context manipulation – variables

$\downarrow$

additional terms for context manipulation – fixed potential

$\downarrow$

bound (using linear programming)

- Now collect constraints and solve LP as before.
- (Rough) SML implementation.

The full analysis

- Have algebraic data types, not just trees.
- Can specify the calculation of potential: depth, total size, mixture
  Bounds w.r.t. total size useful when depth analysis fails.
- Resource polymorphism (different function signatures at different points).

Further work

- Nested types don't behave that well. Have done some work on separating contents and structure
- Inferring the structure of function signatures.
- Reduce complexity of inference.
- Try heap space version.