

Foreword

These are slides* presented by Steffen Jost at
IFL'06 on Wednesday, 6 September, Budapest, Hungary

Further information can be found at my homepage
<http://www.cs.st-andrews.ac.uk/~jost>

Feel free to contact me via email: jost@cs.st-andrews.ac.uk

*Yellow slides were not shown



Analysing HUME

Kevin Hammond

Steffen Jost

Hans-Wolfgang Loidl

University of St Andrews, Scotland

LMU Munich, Bavaria

Budapest, 6 September 2006

The Idea:

Amortised Analysis

Well-known analysis technique used in Complexity Theory

Linear Programming

Well-known efficient technique of solving linear constraints

Combination: Efficient *compile-time* heap analysis for

- first-order functional code (Hofmann & Jost, POPL'03)
- object-oriented imperative code (Java: downcast, inheritance, aliasing)
(Hofmann & Jost, ESOP'06)

To Do: Get life, get real!

The Idea:

Amortised Analysis

Well-known analysis technique used in Complexity Theory

Linear Programming

Well-known efficient technique of solving linear constraints

Combination: Efficient *compile-time* heap analysis for

- first-order functional code (Hofmann & Jost, POPL'03)
- object-oriented imperative code (Java: downcast, inheritance, aliasing)
(Hofmann & Jost, ESOP'06)

To Do: Get life, get real \Rightarrow EmBounded Framework

Analysing HUME

Required extensions over previous work

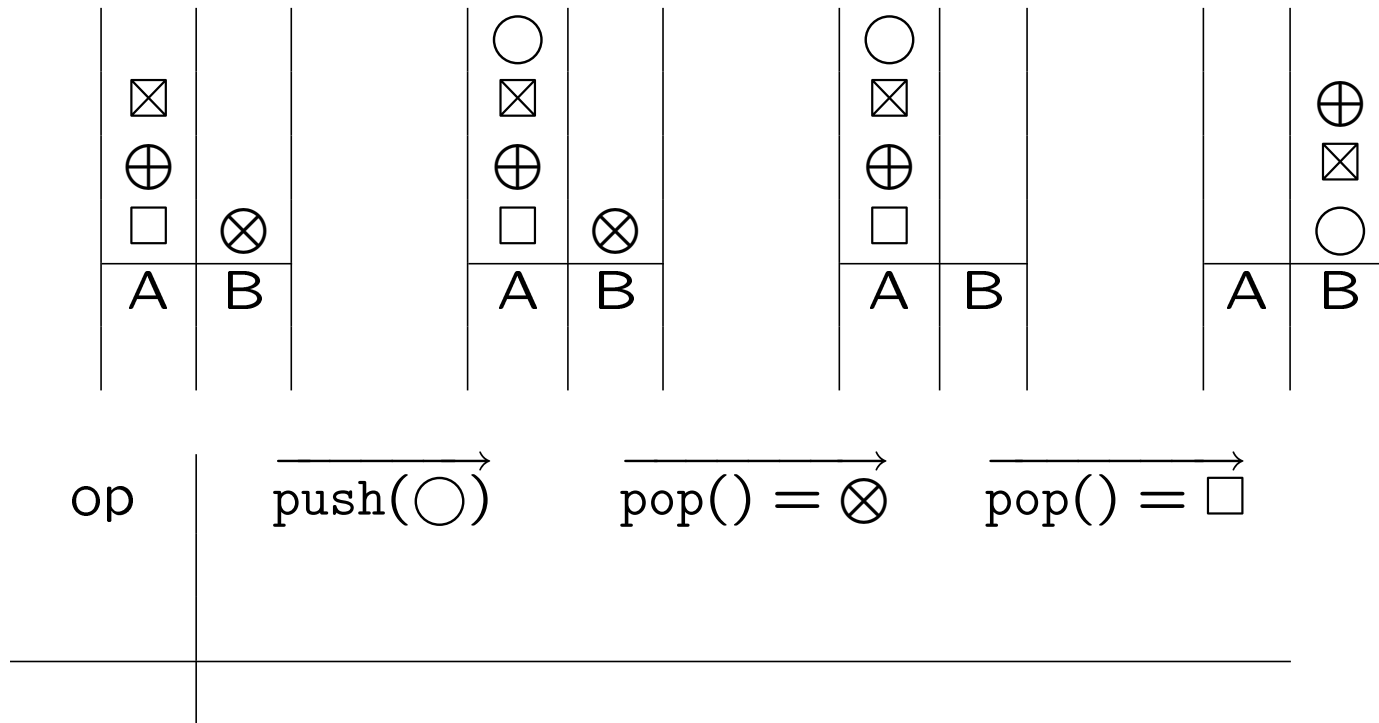
- Higher-order types
- Polymorphism
- Bounding non-terminating computation
- Formal soundness theorem despite loss of simplifications
- Bound Stack space as well as Heap space
- Bound Execution time

carried out within EU FET-IST project [EmBounded](#)
but only *a part* of EmBounded

Amortised Analysis

Example: Simulating queue (FIFO) by two stacks (LIFO)

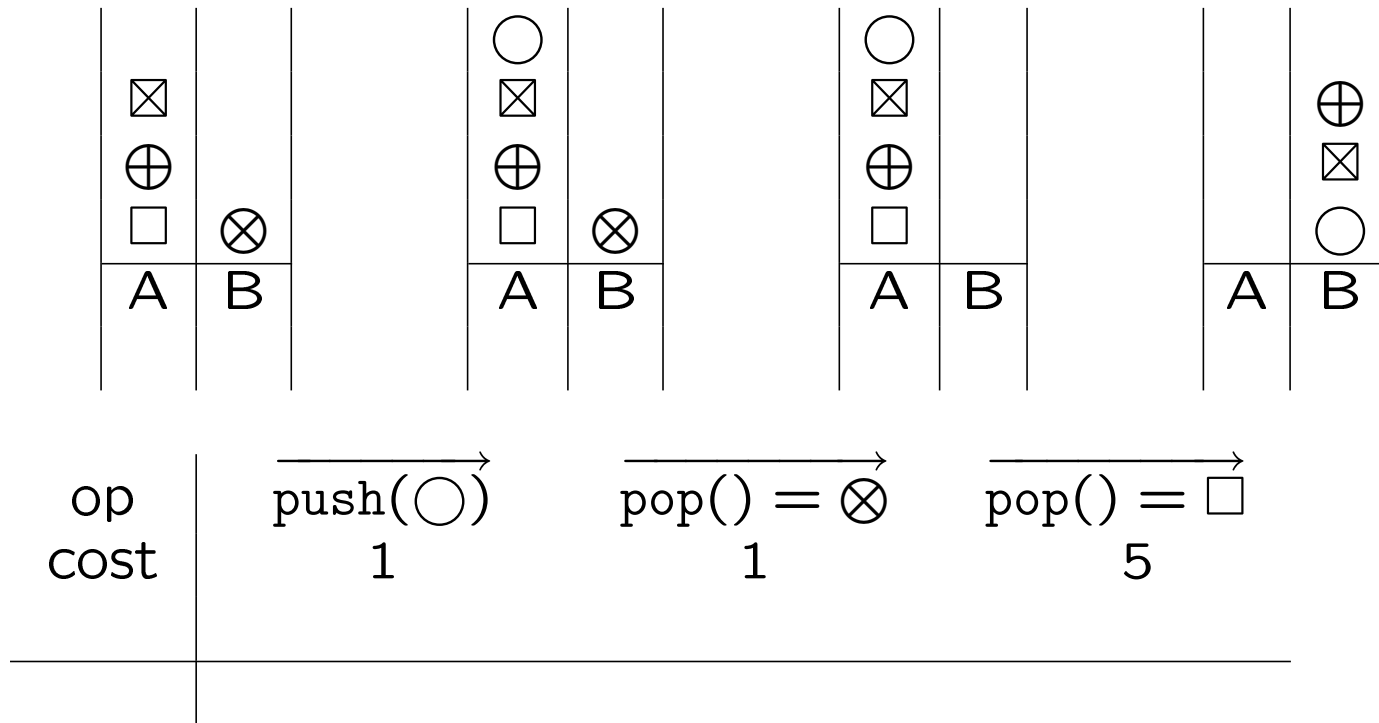
Always push onto A and pop from B



Amortised Analysis

Example: Simulating queue (FIFO) by two stacks (LIFO)

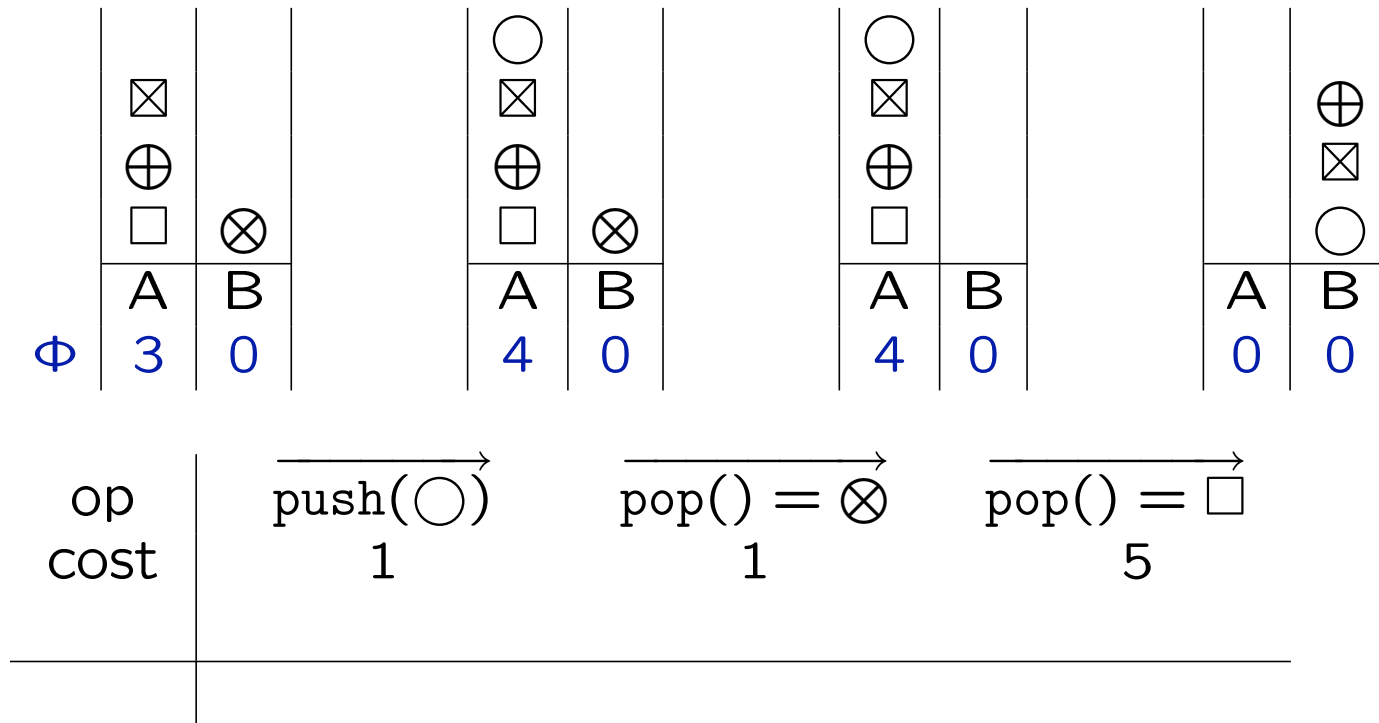
Always push onto A and pop from B



Amortised Analysis

Example: Simulating queue (FIFO) by two stacks (LIFO)

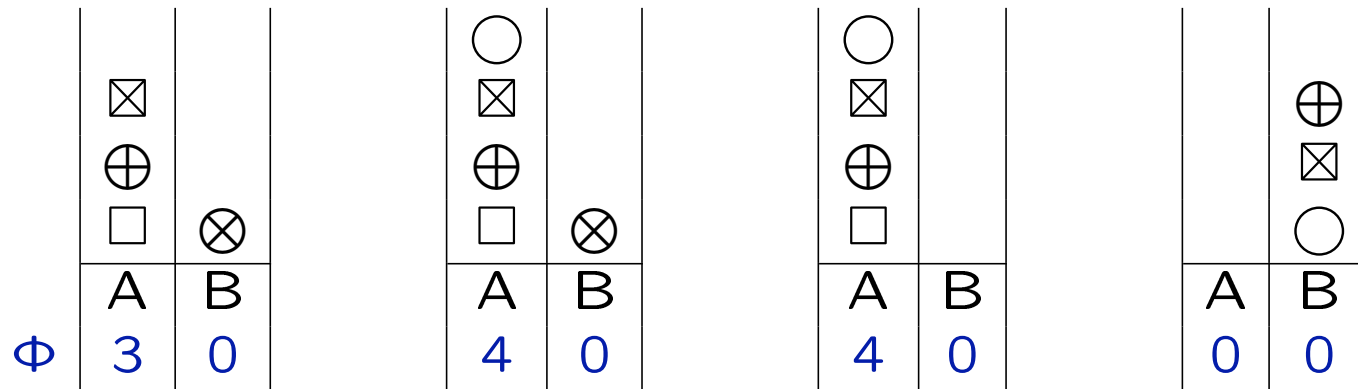
Always push onto A and pop from B



Amortised Analysis

Example: Simulating queue (FIFO) by two stacks (LIFO)

Always push onto A and pop from B



op	$\overrightarrow{\text{push}(\bigcirc)}$	$\overrightarrow{\text{pop}() = \otimes}$	$\overrightarrow{\text{pop}() = \square}$
cost	1	1	5
$\Delta\Phi$	1	0	-4
Σ	2	1	1

Amortised costs are constant as opposed to actual cost!

Automated Analysis of Functional Code: Idea

- Assign potential to data based on type
Type constructors receive weights $(\text{list}(\text{int}, 0), \text{list}(\text{int}, 1), \dots)$
- Abstract from actual values $(\text{list}(\text{int}, x), \text{list}(\text{int}, y), \dots)$
- Gather constraints from type derivation with amortised costs
- Feed constraints to LP solver

Successful heap-space analysis of first-order functional programs
applied in EU FET-IST project [Mobile Resource Guarantees](#)

Automated Analysis of Functional Code: Idea

- Assign potential to data based on type
Type constructors receive weights $(\text{list}(\text{int}, 0), \text{list}(\text{int}, 1), \dots)$
- Abstract from actual values $(\text{list}(\text{int}, x), \text{list}(\text{int}, y), \dots)$
- Gather constraints from type derivation with amortised costs
- Feed constraints to LP solver

Successful heap-space analysis of first-order functional programs
applied in EU FET-IST project [Mobile Resource Guarantees](#)

Automated Analysis of Functional Code: Idea

- Assign potential to data based on type
Type constructors receive weights $(\text{list}(\text{int}, 0), \text{list}(\text{int}, 1), \dots)$
- Abstract from actual values $(\text{list}(\text{int}, x), \text{list}(\text{int}, y), \dots)$
- Gather constraints from type derivation with amortised costs
- Feed constraints to LP solver

Successful heap-space analysis of first-order functional programs
applied in EU FET-IST project [Mobile Resource Guarantees](#)

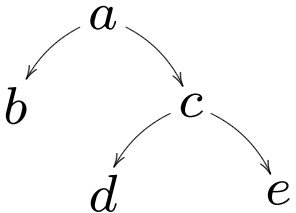
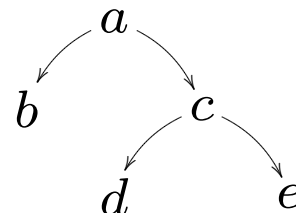
Automated Analysis of Functional Code: Idea

- Assign potential to data based on type
Type constructors receive weights $(\text{list}(\text{int}, 0), \text{list}(\text{int}, 1), \dots)$
- Abstract from actual values $(\text{list}(\text{int}, x), \text{list}(\text{int}, y), \dots)$
- Gather constraints from type derivation with amortised costs
- Feed constraints to LP solver

Successful heap-space analysis of first-order functional programs
applied in EU FET-IST project [Mobile Resource Guarantees](#)

Calculation Example:

type tree= Leaf of char | Node of char*tree*tree

Enriched Type	Instance	Φ
<code>tree[char,0 char,#,#,3]</code>		$3 \cdot 0 + 2 \cdot 3 = 6$
<code>tree[char,4 char,#,#,1]</code>		$3 \cdot 4 + 2 \cdot 1 = 14$

Calculation Example:

type tree= Leaf of char | Node of char*tree*tree

Enriched Type	Instance	Φ
<code>tree[char,0 char,#,#,3]</code>	<pre>graph TD; a --> b; a --> c; c --> d; c --> e;</pre>	$3 \cdot 0 + 2 \cdot 3 = 6$
<code>tree[char,4 char,#,#,1]</code>	<pre>graph TD; a --> c; a --> e; c --> c; c --> e;</pre>	$3 \cdot 4 + 2 \cdot 1 = 14$

- Annotations are linearly distributed in aliased data

Automated Analysis of Functional Code: Result

$$f : \text{list}(\text{list}(\text{int}, 1), 2.3) \xrightarrow{4/6} \text{list}(\text{int}, 5)$$

Evaluating $f([l_1, \dots, l_m])$

- requires at most $4 + 2.3m + 1\sum |l_i|$ extra heap units and
- leaves at least $6 + 5|f(l)|$ unused memory units

Potential of consumed input furnishes upper bound on overall heap-consumption at runtime – without any runtime mechanics!

Annotations are weight factors – *no* reference to length/size
as *opposed* to sized types [Hughes & Pareto '99,'02]

Roadmap

Costed source-level operational semantics

Cost correspondance to target-level semantics

Annotated type system

Soundness wrt to costed source-level operational semantics

Prototype implementation

Certificate generation and verification for target-level code

→ Isabelle

Troubles so far

Higher-order

Determining closure sizes in presence of under application

Solution: Incremental closures

Polymorphism

Resource annotations must be polymorphic too

Solution: Constraint sets become part of types

Exceptions and loss of let-normal form

Causes many cumbersome redundancies

Solution: Alternate syntactic to let having different cost

$$fid\ e_1 \dots e_n \quad \rightsquigarrow \quad LET\ x_1 = e_1\ in\ LET\ \dots\ in\ fid\ x_1 \dots x_n$$

Soundness Theorem (simplified)

If a term types $\Gamma \frac{p}{p'} e : A \mid \Psi$

and evaluates $\mathcal{V}, \eta \vdash e \rightsquigarrow \ell, \eta'$

then for all $P \geq p + \Phi_{\eta}(\mathcal{V}:\Gamma)$

there exist $P' \geq p' + \Phi_{\eta'}(\ell:A)$

such that $\mathcal{V}, \eta \frac{P}{P'} e \rightsquigarrow \ell, \eta'$

Repeat for each resource (time, stack, heap)

Separate theorem dealing with non-termination

Conclusion

- Tedious, but doable
- More examples and implementation are being worked on.
 EmBounded Workshop: Welcome to the lab tomorrow!
- Method easily applicable to other quantitative properties:
 number of calls to certain method, file-handles, etc.
- Interesting to see how amortised analysis will integrate to
 previous HUME analysis

HUME Abstract Syntax

<i>program</i>	$::=$	$decl_1 ; \dots decl_n ;$	$n \geq 1$
<i>decl</i>	$::=$	$box \mid id = expr \mid id \langle match_1 \mid \dots \mid match_n \rangle$	$n \geq 1$
<i>box</i>	$::=$	$box \ id \ (ins) \ (outs) \ fairness \ bmatches \ [handle \ cmatches]$	
<i>ins,outs</i>	$::=$	$\langle id_1, \dots, id_n \rangle$	$n \geq 0$
<i>fairness</i>	$::=$	$fair \mid unfair$	
<i>bmatches</i>	$::=$	$\langle bmatch_1 \mid \dots \mid bmatch_n \rangle$	$n \geq 1$
<i>bmatch</i>	$::=$	$\langle bpat_1, \dots, bpat_n \rangle \rightarrow expr$	$n \geq 1$
<i>cmatches</i>	$::=$	$\langle cmatch_1 \mid \dots \mid cmatch_n \rangle$	$n \geq 1$
<i>cmatch</i>	$::=$	$exn \ pat \rightarrow exnexpr$	
<i>bpat</i>	$::=$	$pat \mid * \mid _*$	
<i>pat</i>	$::=$	$int \mid float \mid char \mid bool \mid string \mid _ \mid var$ $\mid con \ pat_1 \ \dots \ pat_n$ $\mid (\ pat_1 \ , \ \dots \ , \ pat_n \)$	$n \geq 0$ $n \geq 2$

HUME Abstract Syntax (cont'd)

```

expr ::= int | float | char | bool | string | *
      | var expr1 ... exprn n ≥ 0
      | id expr1 ... exprn n ≥ 0
      | con expr1 ... exprn n ≥ 0
      | ( expr1 , ... , exprn ) n ≥ 2
      | if expr1 then expr2 else expr3
      | case expr of { match1 | ... | matchn } n ≥ 1
      | let { vdecl1 , ... , vdecln } in expr n ≥ 1
      | expr within int time raise exn exnexpr
      | expr within int stack raise exn exnexpr
      | expr within int heap raise exn exnexpr
      | raise exn exnexpr

match ::= pat -> expr
vdecl ::= var = expr

```

HUME Abstract Syntax (cont'd)

```

exnexpr ::= int | float | char | bool | string | *
          | var expr1 ... exprn n ≥ 0
          | id expr1 ... exprn n ≥ 0
          | con exnexpr1 ... exnexprn n ≥ 0
          | ( exnexpr1 , ... , exnexprn ) n ≥ 2
          | if exnexpr1 then exnexpr2 else exnexpr3
          | case exnexpr of { exnmatch1 | ... | exnmatchn } n ≥ 1
          | let { exnvdecl1 , ... , exnvdecln } in exnexpr n ≥ 1
          | expr within int time raise exn exnexpr
          | expr within int stack raise exn exnexpr
          | expr within int heap raise exn exnexpr
          | raise exn exnexpr

exnmatch ::= pat -> exnexpr
exnvdecl ::= var = exnexpr

```

<i>expr</i> :	black & green
<i>exnexpr</i> :	black & red