

Foreword

These are slides* presented by Steffen Jost at [EmBounded Workshop 2006](#) on Thursday, 7 September, Budapest, Hungary

It is expected that the reader has attended my talk at IFL2006 on the previous day!

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 text and yellow slides were not shown!



Analysing HUME - Practical

Steffen Jost

University of St Andrews, Scotland

Budapest, 7 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

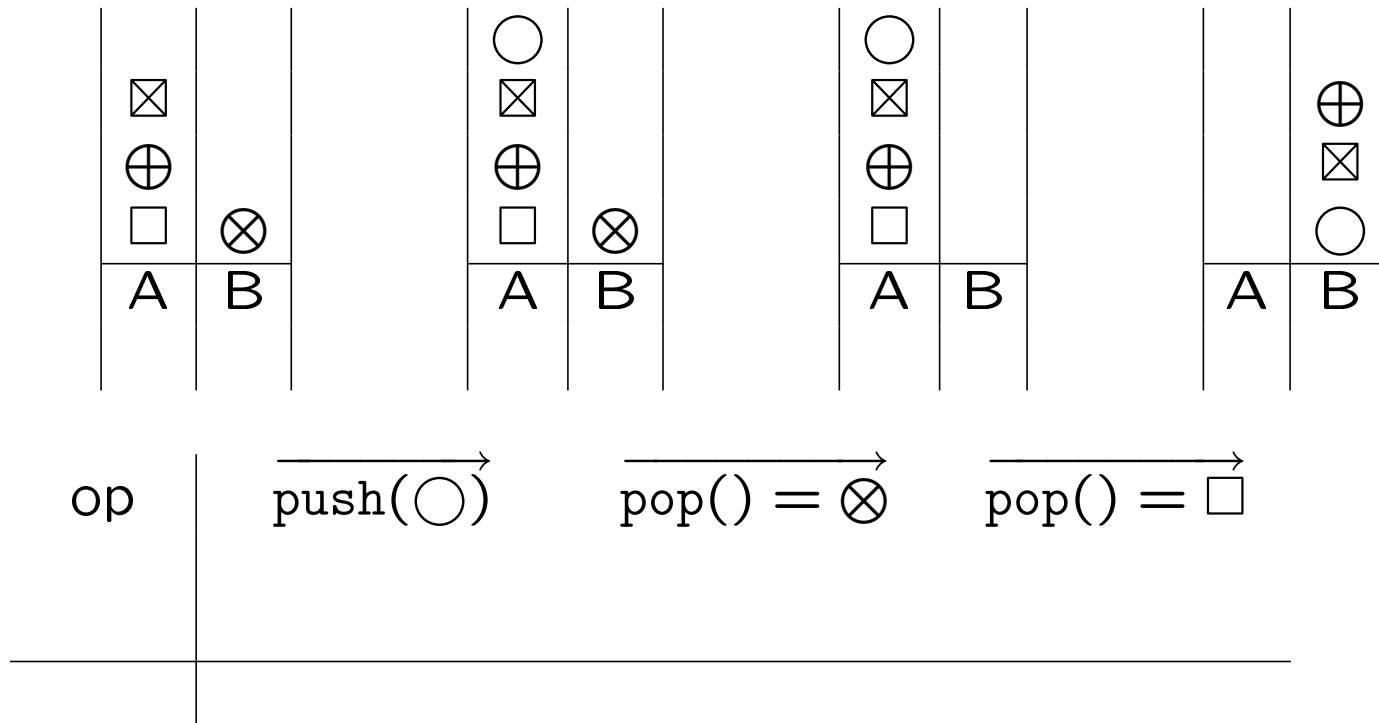
- first-order functional code (Hofmann & Jost, POPL'03)
- object-oriented imperative code (Java: downcast, inheritance, aliasing)
(Hofmann & Jost, ESOP'06)

To Do: Get life, get real ⇒ **EmBounded** Framework

Amortised Analysis

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

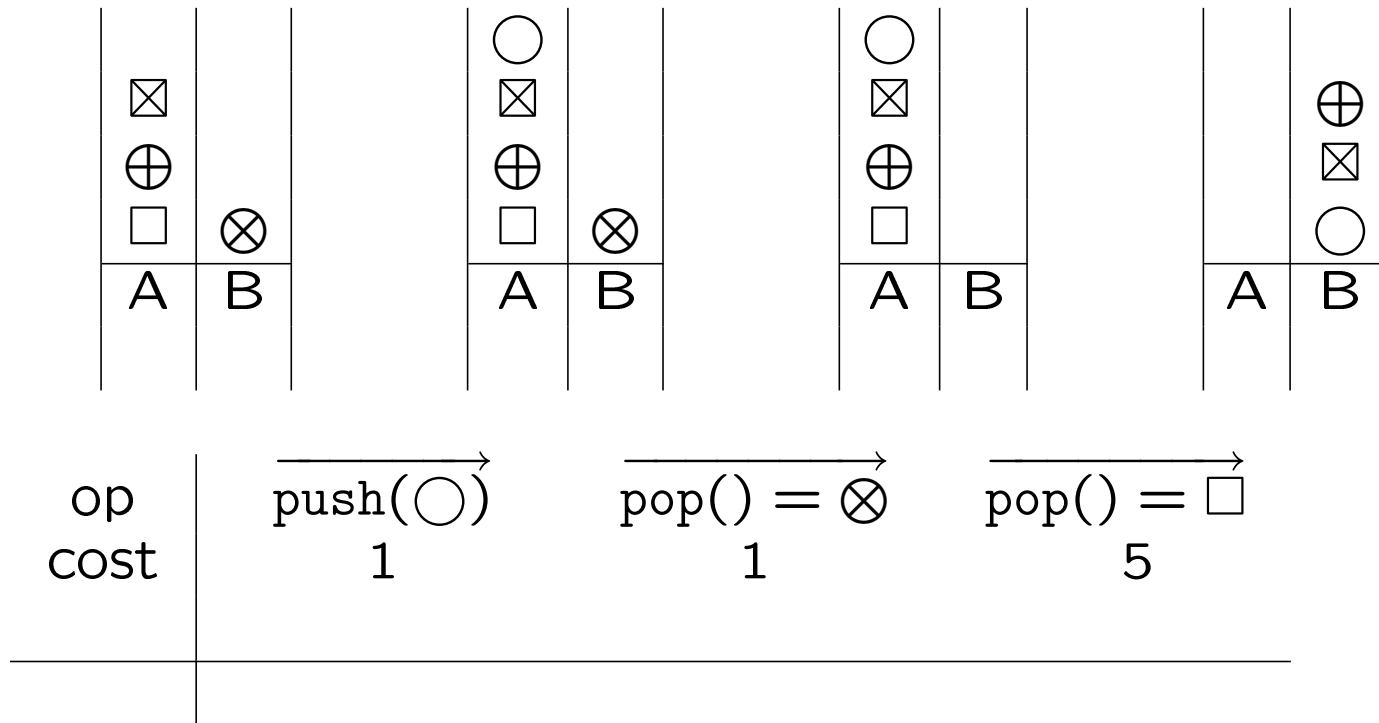
Always push onto A and pop from B



Amortised Analysis

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

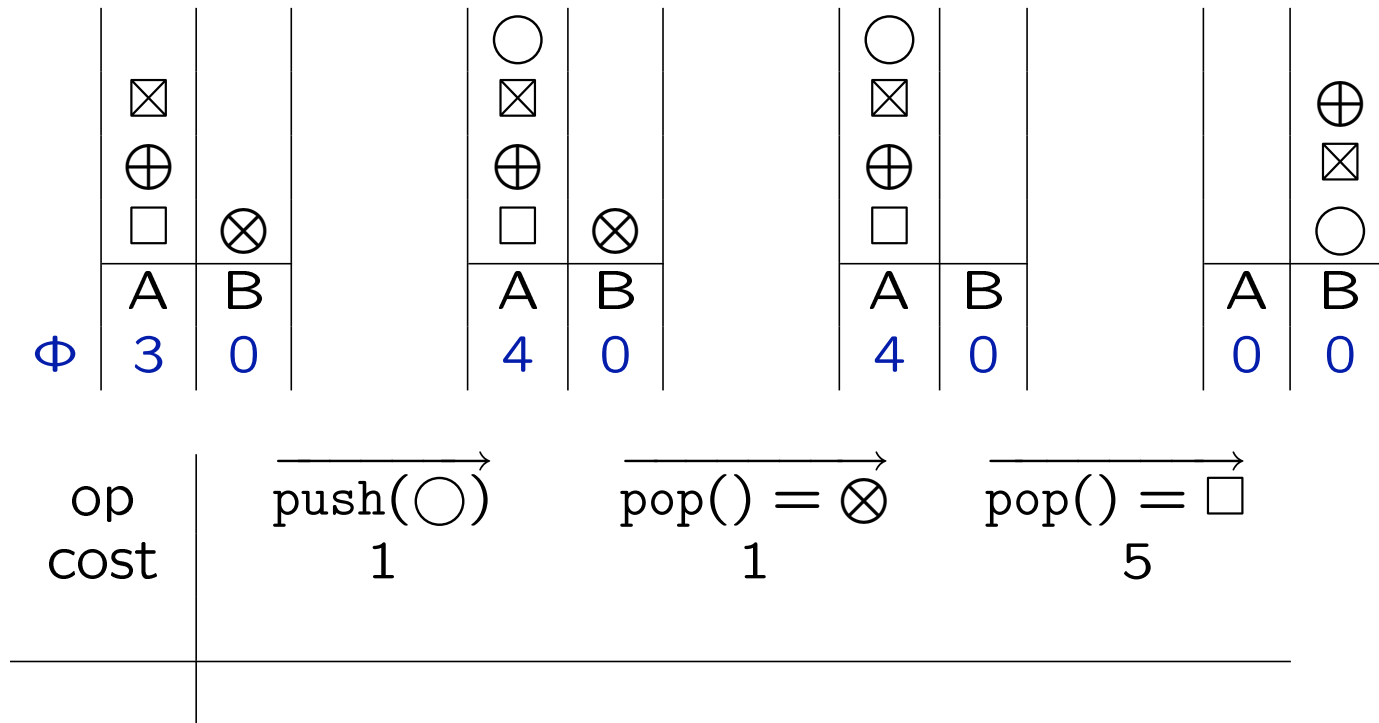
Always push onto A and pop from B



Amortised Analysis

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

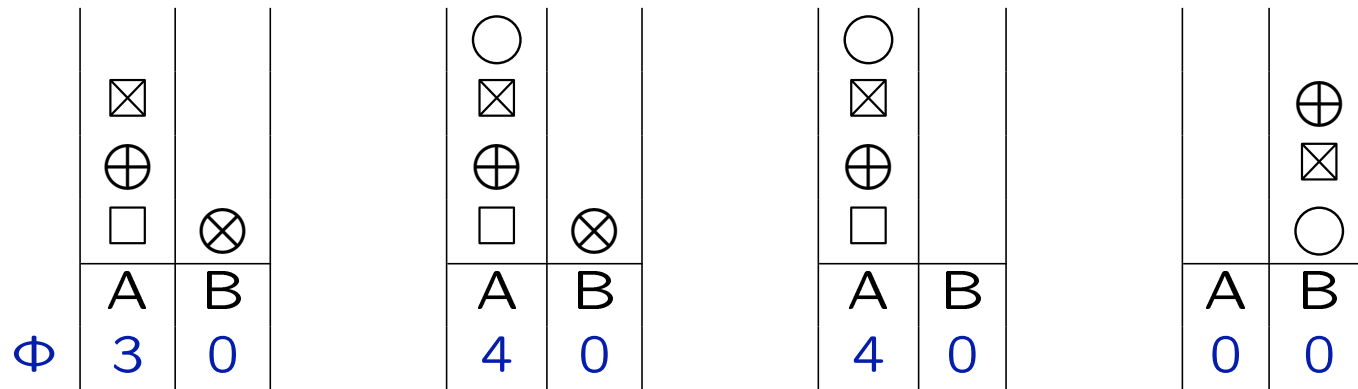
Always push onto A and pop from B



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](#)

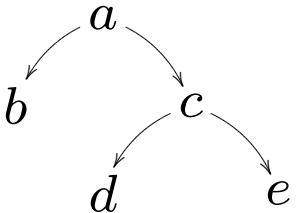
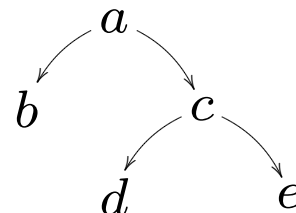
Automated Analysis of Functional Code: Idea

- 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]

Informal calculation:

$\text{rev_aux} : \text{list}(\text{int }) , \text{list}(\text{int }) \longrightarrow \text{list}(\text{int })$

$$\begin{array}{l} \text{rev_aux}([], \text{acc}) = \text{acc} \\ \text{rev_aux}(h :: t, \text{acc}) = \text{rev_aux}(t, h :: \text{acc}) \end{array} \left| \right.$$

- In ASCII we write $A, 3 \rightarrow B, 4$ for type $A \xrightarrow{3/4} B$

Informal calculation:

$\text{rev_aux} : \text{list}(\text{int}, u), \text{list}(\text{int}, v), x \longrightarrow \text{list}(\text{int}, w), y$

$$\begin{array}{l} \text{rev_aux}([], acc) = acc \\ \text{rev_aux}(h :: t, acc) = \text{rev_aux}(t, h :: acc) \end{array} \left| \begin{array}{l} \{x \geq y, v \geq w\} \\ \{x + u \geq \zeta + p, \\ p \geq x, p \geq x - y + y\} \end{array} \right.$$

- In ASCII we write $A, 3 \rightarrow B, 4$ for type $A \xrightarrow{3/4} B$
- Enrich type system with resource variables

Informal calculation:

$\text{rev_aux} : \text{list}(\text{int}, u), \text{list}(\text{int}, v), x \longrightarrow \text{list}(\text{int}, w), y$

$$\begin{array}{l} \text{rev_aux}([], acc) = acc \\ \text{rev_aux}(h :: t, acc) = \text{rev_aux}(t, h :: acc) \end{array} \left| \begin{array}{l} \{x \geq y, v \geq w\} \\ \{x + u - \zeta \geq x, \\ x + u - \zeta \geq x - y + y\} \end{array} \right.$$

- In ASCII we write $A, 3 \multimap B, 4$ for type $A \xrightarrow{3/4} B$
- Enrich type system with resource variables
- Gather linear constraints

Informal calculation:

$\text{rev_aux} : \text{list}(\text{int}, u), \text{list}(\text{int}, v), x \longrightarrow \text{list}(\text{int}, w), y$

$$\begin{array}{l} \text{rev_aux}([], acc) = acc \\ \text{rev_aux}(h :: t, acc) = \text{rev_aux}(t, h :: acc) \end{array} \left| \begin{array}{l} \{x \geq y, v \geq w\} \\ \{x + u - \zeta \geq x, \\ x + u - \zeta \geq x - y + y\} \end{array} \right.$$

$x = 0, y = 0, u = 1, v = 0, w = 0; \text{ for } \zeta = 1$

- In ASCII we write $A, 3 \rightarrow B, 4$ for type $A \xrightarrow{3/4} B$
- Enrich type system with resource variables
- Gather linear constraints
- Solve constraints by standard methods

Informal calculation:

$\text{rev_aux} : \text{list}(\text{int}, 1), \text{list}(\text{int}, 0), 0 \longrightarrow \text{list}(\text{int}, 0), 0$

$$\begin{array}{l} \text{rev_aux}([], acc) = acc \\ \text{rev_aux}(h :: t, acc) = \text{rev_aux}(t, h :: acc) \end{array} \left| \begin{array}{l} \{0 \geq 0, 0 \geq 0\} \\ \{0 + 1 - 1 \geq 0\} \\ 0 + 1 - 1 \geq 0 - 0 + 0 \} \end{array} \right.$$

$$x = 0, y = 0, u = 1, v = 0, w = 0; \text{ for } \zeta = 1$$

- In ASCII we write $A, 3 \rightarrow B, 4$ for type $A \xrightarrow{3/4} B$
- Enrich type system with resource variables
- Gather linear constraints
- Solve constraints by standard methods

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>	$::=$	$int \mid float \mid char \mid bool \mid string \mid _ \mid var$ $\mid \text{con } pat_1 \ \dots \ pat_n$ $\mid (\ pat_1 \ , \ \dots \ , \ pat_n \)$	$n \geq 0$ $n \geq 2$

HUME Abstract Syntax (cont'd)

<i>expr</i>	<code>::=</code>	<code>int float char bool string *</code>	
		<code>var <i>expr</i>₁ ... <i>expr</i>_{<i>n</i>}</code>	<i>n</i> ≥ 0
		<code>id <i>expr</i>₁ ... <i>expr</i>_{<i>n</i>}</code>	<i>n</i> ≥ 0
		<code>con <i>expr</i>₁ ... <i>expr</i>_{<i>n</i>}</code>	<i>n</i> ≥ 0
		<code>(<i>expr</i>₁ , ... , <i>expr</i>_{<i>n</i>})</code>	<i>n</i> ≥ 2
		<code>if <i>expr</i>₁ then <i>expr</i>₂ else <i>expr</i>₃</code>	
		<code>case <i>expr</i> of < <i>match</i>₁ ... <i>match</i>_{<i>n</i>} ></code>	<i>n</i> ≥ 1
		<code>let < <i>vdecl</i>₁ , ... , <i>vdecl</i>_{<i>n</i>} > in <i>expr</i></code>	<i>n</i> ≥ 1
		<code><i>expr</i> within int time raise <i>exn</i> <i>exnexpr</i></code>	
		<code><i>expr</i> within int stack raise <i>exn</i> <i>exnexpr</i></code>	
		<code><i>expr</i> within int heap raise <i>exn</i> <i>exnexpr</i></code>	
		<code>raise <i>exn</i> <i>exnexpr</i></code>	
<i>match</i>	<code>::=</code>	<code><i>pat</i> -> <i>expr</i></code>	
<i>vdecl</i>	<code>::=</code>	<code>var = <i>expr</i></code>	

HUME Abstract Syntax (cont'd)

$exnexpr ::= int \mid float \mid char \mid bool \mid string \mid *$
 $\quad \mid \text{var } expr_1 \cdots expr_n \quad n \geq 0$
 $\quad \mid id \ expr_1 \cdots expr_n \quad n \geq 0$
 $\quad \mid \text{con } exnexpr_1 \cdots exnexpr_n \quad n \geq 0$
 $\quad \mid (\ expr_1 , \dots , expr_n) \quad n \geq 2$
 $\quad \mid \text{if } exnexpr_1 \text{ then } exnexpr_2 \text{ else } exnexpr_3$
 $\quad \mid \text{case } exnexpr \text{ of } \langle exnmatch_1 \mid \dots \mid exnmatch_n \rangle \quad n \geq 1$
 $\quad \mid \text{let } \langle exnvdecl_1 , \dots , exnvdecl_n \rangle \text{ in } exnexpr \quad n \geq 1$
 $\quad \mid \text{expr within int time raise exn } exnexpr$
 $\quad \mid \text{expr within int stack raise exn } exnexpr$
 $\quad \mid \text{expr within int heap raise exn } exnexpr$
 $\quad \mid \text{raise exn } exnexpr$

$exnmatch ::= pat \rightarrow exnexpr$
 $exnvdecl ::= \text{var } = exnexpr$

$expr$:	black & green
$exnexpr$:	black & yellow

Conditional

Annotated Type Rule

$$\frac{\Gamma \frac{p}{p'} \frac{m}{m'} e_t : A \mid \Phi \quad \Gamma \frac{p}{p'} \frac{m}{m'} e_f : A \mid \Psi}{\Gamma, x:\text{bool} \frac{p-1}{p'} \frac{m}{m'} \text{if } x \text{ then } e_t \text{ else } e_f : A \mid \Phi \cup \Psi} \text{(Conditional)}$$

Annotated Operational Semantics

$$\frac{\eta(\mathcal{V}(x)) = (\text{bool}, \text{tt}) \quad \mathcal{V}, \eta \frac{p+1}{p'} \frac{m}{m'} e_1 \rightsquigarrow \ell', \eta'}{\mathcal{V}, \eta \frac{p}{p'} \frac{m}{m'} \text{if } x \text{ then } e_1 \text{ else } e_2 \rightsquigarrow \ell', \eta'} \text{(Conditional True)}$$

$$\frac{\eta(\mathcal{V}(x)) = (\text{bool}, \text{ff}) \quad \mathcal{V}, \eta \frac{p+1}{p'} \frac{m}{m'} e_2 \rightsquigarrow \ell', \eta'}{\mathcal{V}, \eta \frac{p}{p'} \frac{m}{m'} \text{if } x \text{ then } e_1 \text{ else } e_2 \rightsquigarrow \ell', \eta'} \text{(Conditional False)}$$

Reading Rules

$$\frac{}{\mathcal{V}, \eta \mid \frac{p'+1}{p'} \mid \frac{m}{m-2} e \rightsquigarrow \ell, \eta'}$$

is an abbreviation for

$$\frac{\begin{array}{l} p = p' + 1 \\ m = m' + 2 \end{array} \quad \begin{array}{l} p \in \mathbb{N} \\ m \in \mathbb{N} \end{array} \quad \begin{array}{l} p' \in \mathbb{N} \\ m' \in \mathbb{N} \end{array}}{\mathcal{V}, \eta \mid \frac{p}{p'} \mid \frac{m}{m'} e \rightsquigarrow \ell, \eta'}$$

Application

$$\frac{\Sigma^{\mathcal{F}}(fid) = \left(-; -; A_1, \dots, A_a \frac{p}{p'} \frac{m}{m'} C \right) \quad k \geq 1 \quad k = a}{y_1:A_1, \dots, y_k:A_k \mid \frac{p}{p'+k} \frac{m}{m'} fid \ y_1 \cdots y_k : C \mid \emptyset} \text{ (App)}$$

$$\Sigma^{\mathcal{F}}(fid) = \left(-; -; A_1, \dots, A_a \frac{q}{q'} \frac{n}{n'} C \right) \quad k \geq 1 \quad k < a$$

$$\Phi = \{p = 0, p' = k - 1, m = \text{Size}(\text{Closure}_k), m' = 0\}$$

$$\frac{y_1:A_1, \dots, y_k:A_k \mid \frac{p}{p'} \frac{m}{m'} fid \ y_1 \cdots y_k : A_{k+1}, \dots, A_a \frac{q+k}{q'} \frac{n}{n'} C \mid \Phi}{\text{ (Under App)}}$$

$$\Sigma^{\mathcal{F}}(fid) = \left(-; -; A_1, \dots, A_a \frac{q}{q'} \frac{n}{n'} C \right) \quad a \geq 1 \quad k > a$$

$$y_1:A_1, \dots, y_a:A_a \mid \frac{p}{p'} \frac{m}{m'} fid \ y_1 \cdots y_a : C \mid \Phi$$

$$\hat{z} \text{ is fresh} \quad C = B_1, \dots, B_b \frac{q}{q'} \frac{n}{n'} E$$

$$\hat{z}:C, y_{a+1}:A_{a+1}, \dots, y_k:A_k \mid \frac{p'}{p''} \frac{m'}{m''} \hat{z} \ y_{a+1} \cdots y_k : E \mid \Psi$$

$$\frac{y_1:A_1, \dots, y_k:A_k \mid \frac{p}{p''} \frac{m}{m''} fid \ y_1 \cdots y_k : E \mid \Phi \cup \Psi}{\text{ (Over App)}}$$

Sized Types versus Amortised Analysis

Example: List splitting (as in Quick-Sort)

```
split x [] = ([], [])
split x (y:ys) = let (l,r) = split x ys in
                  if x > y then (l,y:r) else (y:l,r)
```

ST $\text{Nat} \rightarrow \text{List}_n(\text{Nat}) \rightarrow \text{List}_n(\text{Nat}) \times \text{List}_n(\text{Nat})$

AA $\text{Nat} \xrightarrow{0/0} \text{List}(\text{Nat}, 1) \xrightarrow{0/0} \text{List}(\text{Nat}, 1) \times \text{List}(\text{Nat}, 1)$

However, non-linear consumption is generally not treatable by **AA**, e.g. vector multiplication, which can be treated by **ST**.

Remaining Talk

The talk then continued with practical exercises after the talk of Hans-Wolfgang Loidl.

An earlier but very stable implementation and example programs can be download from my (Steffen Jost) homepage, allowing first-hand experience of the automated static analysis.