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 attened 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

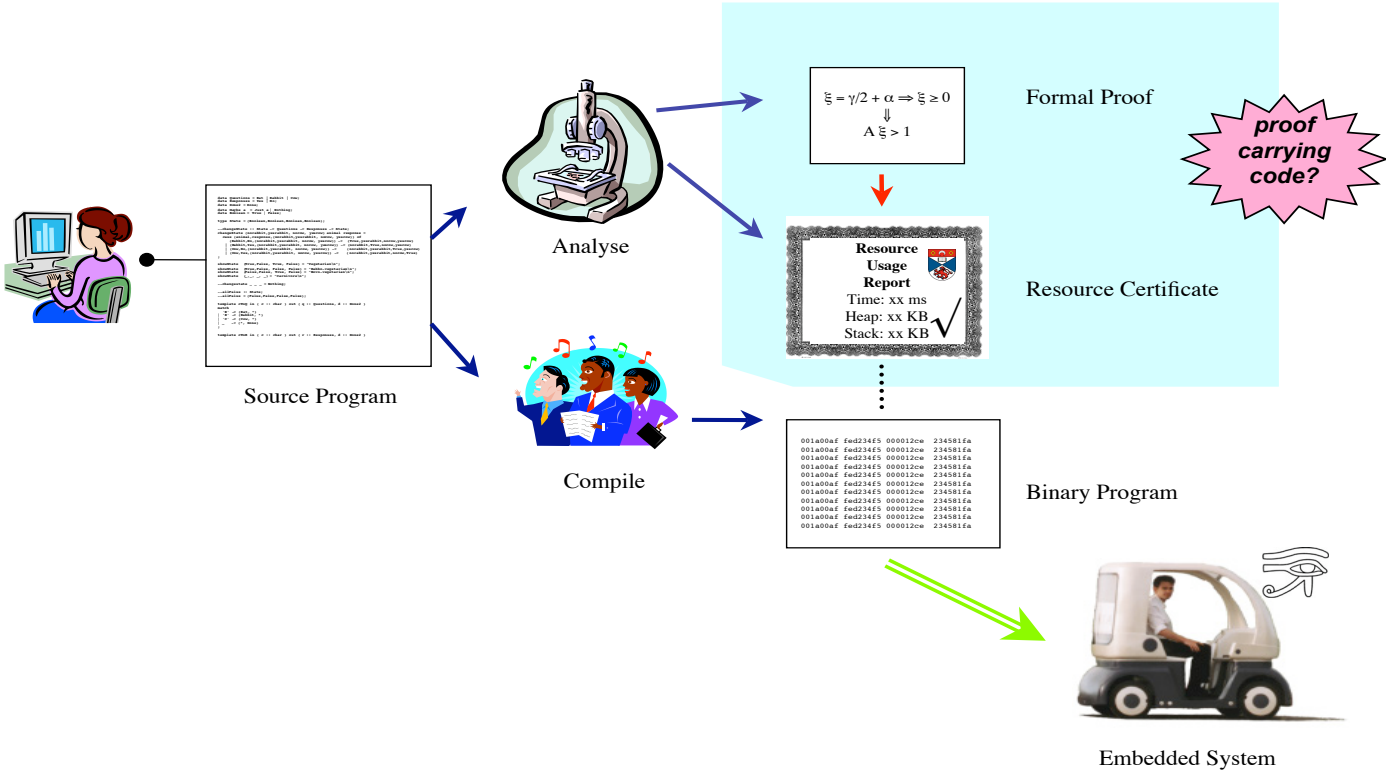
Well-known efficient technique of solving linear constraints

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 $\Rightarrow$ EmBounded Framework

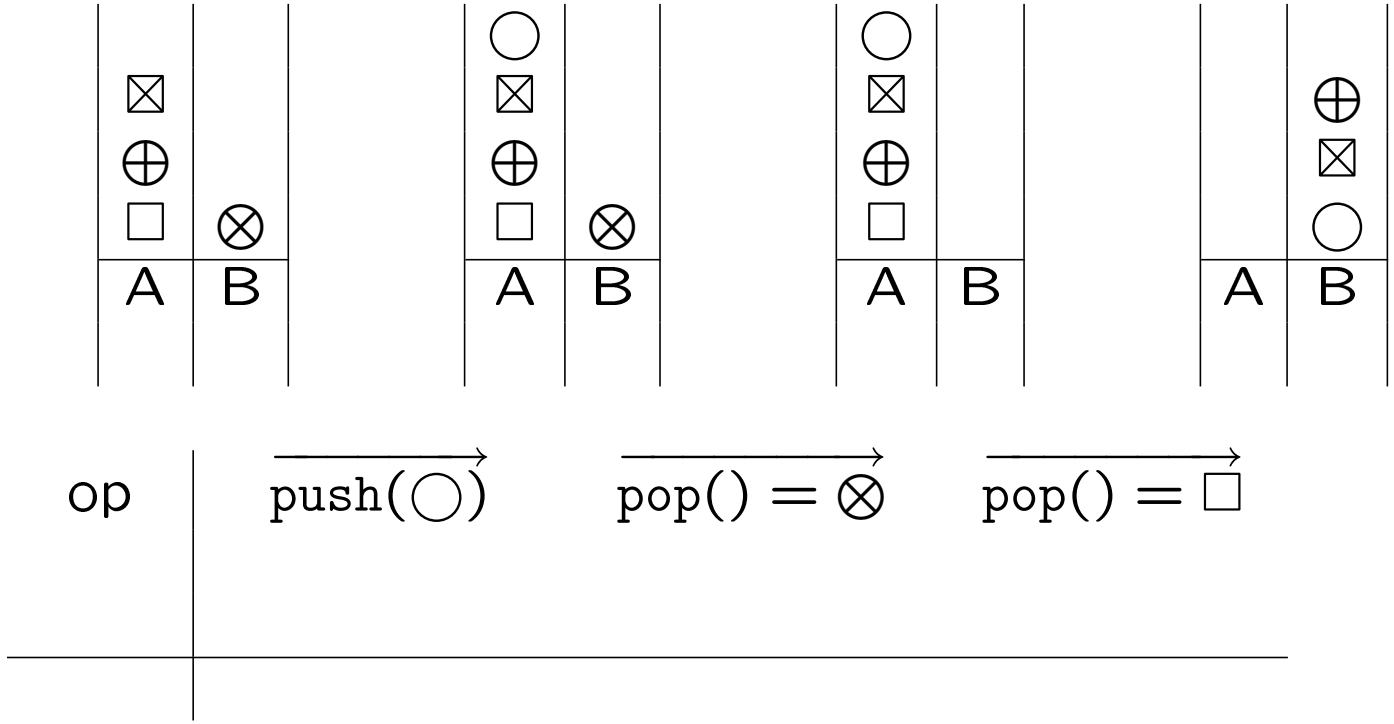
EmBounded Vision



Amortised Analysis

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

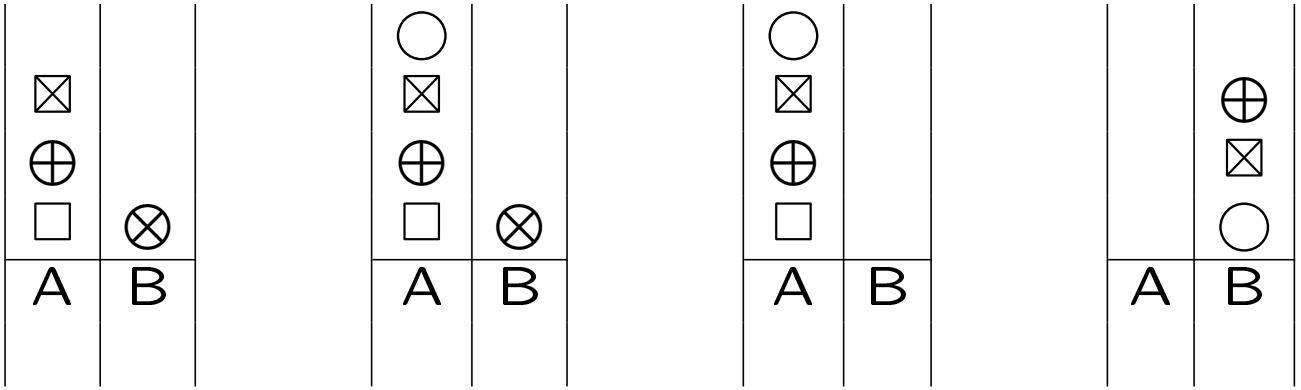
Always push onto A and pop from B



Amortised Analysis

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

Always push onto A and pop from B

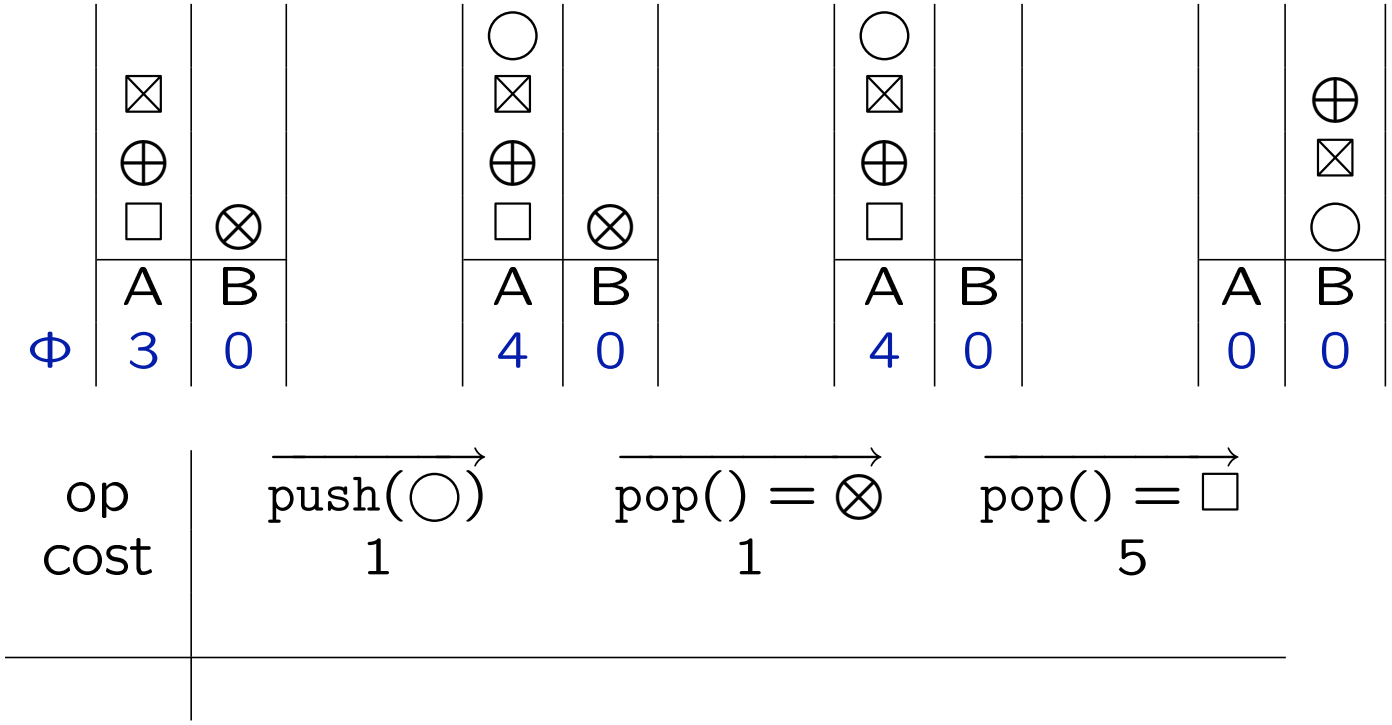


op	$\xrightarrow{\texttt{push}(\bigcirc)}$	$\xrightarrow{\texttt{pop}() = \otimes}$	$\xrightarrow{\texttt{pop}() = \square}$
cost	1	1	5

Amortised Analysis

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

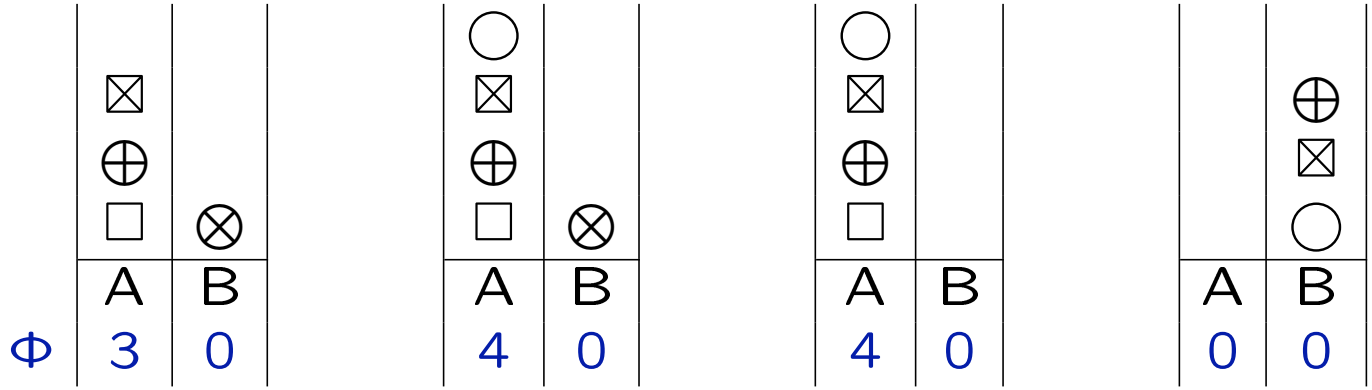
Always push onto A and pop from B



Amortised Analysis

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

Always push onto A and pop from B



op	$\xrightarrow{\texttt{push}(\bigcirc)}$	$\xrightarrow{\texttt{pop}() = \otimes}$	$\xrightarrow{\texttt{pop}() = \square}$
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 $(\mathsf{list}(\mathsf{int}, 0), \mathsf{list}(\mathsf{int}, 1), \ldots)$
- Abstract from actual values $(\mathsf{list}(\mathsf{int}, x), \mathsf{list}(\mathsf{int}, y), \ldots)$
- 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 $(\mathsf{list}(\mathsf{int}, 0), \mathsf{list}(\mathsf{int}, 1), \ldots)$
- Abstract from actual values $(\mathsf{list}(\mathsf{int}, x), \mathsf{list}(\mathsf{int}, y), \ldots)$
- 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 $(\mathsf{list}(\mathsf{int}, 0), \mathsf{list}(\mathsf{int}, 1), \ldots)$
- Abstract from actual values $(\mathsf{list}(\mathsf{int}, x), \mathsf{list}(\mathsf{int}, y), \ldots)$
- 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

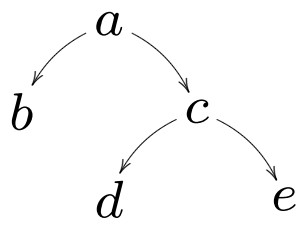
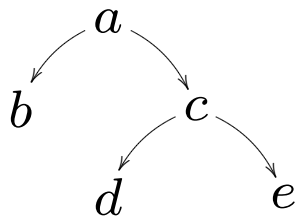
Automated Analysis of Functional Code: Idea

- Assign potential to data based on type
  Type constructors receive weights $(\mathsf{list}(\mathsf{int}, 0), \mathsf{list}(\mathsf{int}, 1), \ldots)$
- Abstract from actual values $(\mathsf{list}(\mathsf{int}, x), \mathsf{list}(\mathsf{int}, y), \ldots)$
- 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

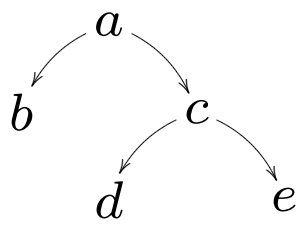
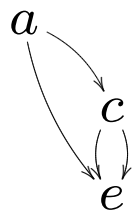
Calculation Example:

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

Enriched Type	Instance	Φ
`tree[char,0|char,#,#,3]`		$3 \cdot 0 + 2 \cdot 3 = 6$
`tree[char,4|char,#,#,1]`		$3 \cdot 4 + 2 \cdot 1 = 14$

Calculation Example:

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

Enriched Type	Instance	Φ
`tree[char,0|char,#,#,3]`		$3 \cdot 0 + 2 \cdot 3 = 6$
`tree[char,4|char,#,#,1]`		$3 \cdot 4 + 2 \cdot 1 = 14$

- Annotations are linearly distributed in aliased data

Automated Analysis of Functional Code: Result

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

Evaluating $\texttt{f}\Big([l_1, \ldots, l_m]\Big)$

- requires at most $4 + 2.3m + 1\Sigma|l_i|$ extra heap units and
- leaves at least $6 + 5|\texttt{f}(l)|$ 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:

$\texttt{rev\_aux} : \mathsf{list(int\ \ )}, \mathsf{list(int\ \ )} \longrightarrow \mathsf{list(int\ \ )}$

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

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

Informal calculation:

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

$$\begin{array}{l|l} \texttt{rev\_aux}([\,], acc) = acc & \{x \geq y\,,\ \ v \geq w\} \\ \texttt{rev\_aux}(h :: t, acc) = \texttt{rev\_aux}(t, h :: acc) & \{x + u \geq \zeta + p\,, \\ & \ p \geq x\,,\, p \geq x - y + y\ \} \end{array}$$

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

Informal calculation:

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

$$\begin{array}{l|l} \texttt{rev\_aux}([\,], acc) = acc & \{x \geq y\,,\ \ v \geq w\} \\ \texttt{rev\_aux}(h :: t, acc) = \texttt{rev\_aux}(t, h :: acc) & \{x + u - \zeta \geq x\,, \\ & \ \ x + u - \zeta \geq x - y + y\} \end{array}$$

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

Informal calculation:

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

$$\begin{array}{l|l} \texttt{rev\_aux}([\,], acc) = acc & \{x \geq y\,,\ \ v \geq w\} \\ \texttt{rev\_aux}(h :: t, acc) = \texttt{rev\_aux}(t, h :: acc) & \{x + u - \zeta \geq x\,, \\ & \ \ x + u - \zeta \geq x - y + y\} \end{array}$$

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

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

Informal calculation:

$\texttt{rev\_aux} : \mathsf{list}(\mathsf{int},1), \mathsf{list}(\mathsf{int},0),0 \longrightarrow \mathsf{list}(\mathsf{int},0),0$

$$\begin{array}{l|l} \texttt{rev\_aux}([\,], acc) = acc & \{0 \geq 0\,,\ \ 0 \geq 0\} \\ \texttt{rev\_aux}(h :: t, acc) = \texttt{rev\_aux}(t, h :: acc) & \{0+1-1 \geq 0 \\ & \ \ 0+1-1 \geq 0-0+0\} \end{array}$$

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

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

HUME Abstract Syntax

program	::=	decl_1 ; $\cdots$ decl_n ;	$n \geq 1$
decl	::=	*box* | id = *expr* | id ⟨ match_1 | $\cdots$ | match_n ⟩	$n \geq 1$
box	::=	`box` id (*ins*) (*outs*) *fairness bmatches* [`handle` *cmatches*]	
ins,outs	::=	⟨ $\mathrm{id}_1, \ldots, \mathrm{id}_n$ ⟩	$n \geq 0$
fairness	::=	`fair` | `unfair`	
bmatches	::=	⟨ bmatch_1 | $\cdots$ | bmatch_n ⟩	$n \geq 1$
bmatch	::=	⟨ bpat_1 , $\ldots$, bpat_n ⟩ -> *expr*	$n \geq 1$
cmatches	::=	⟨ cmatch_1 | $\cdots$ | cmatch_n ⟩	$n \geq 1$
cmatch	::=	exn *pat* -> *exnexpr*	
bpat	::=	*pat* | * | _*	
pat	::=	int | float | char | bool | string | _ | var	
		| con pat_1 $\cdots$ pat_n	$n \geq 0$
		| (pat_1 , $\ldots$, pat_n)	$n \geq 2$

HUME Abstract Syntax (cont'd)

$$
\begin{array}{lcll}
\textit{expr} & ::= & \text{int} \mid \text{float} \mid \text{char} \mid \text{bool} \mid \text{string} \mid * & \\
& \mid & \text{var}\ \textit{expr}_1\ \cdots\ \textit{expr}_n & n \geq 0 \\
& \mid & \text{id}\ \textit{expr}_1\ \cdots\ \textit{expr}_n & n \geq 0 \\
& \mid & \text{con}\ \textit{expr}_1\ \cdots\ \textit{expr}_n & n \geq 0 \\
& \mid & (\ \textit{expr}_1\ ,\ \ldots\ ,\ \textit{expr}_n\ ) & n \geq 2 \\
& \mid & \texttt{if}\ \textit{expr}_1\ \texttt{then}\ \textit{expr}_2\ \texttt{else}\ \textit{expr}_3 & \\
& \mid & \texttt{case}\ \textit{expr}\ \texttt{of}\ \langle\ \textit{match}_1\ \mid\ \cdots\ \mid\ \textit{match}_n\ \rangle & n \geq 1 \\
& \mid & \texttt{let}\ \langle\ \textit{vdecl}_1\ ,\ \ldots\ ,\ \textit{vdecl}_n\ \rangle\ \texttt{in}\ \textit{expr} & n \geq 1 \\
& \mid & \textit{expr}\ \texttt{within}\ \text{int}\ \texttt{time}\ \texttt{raise}\ \text{exn}\ \textit{exnexpr} & \\
& \mid & \textit{expr}\ \texttt{within}\ \text{int}\ \texttt{stack}\ \texttt{raise}\ \text{exn}\ \textit{exnexpr} & \\
& \mid & \textit{expr}\ \texttt{within}\ \text{int}\ \texttt{heap}\ \texttt{raise}\ \text{exn}\ \textit{exnexpr} & \\
& \mid & \texttt{raise}\ \text{exn}\ \textit{exnexpr} & \\
\textit{match} & ::= & \textit{pat}\ \texttt{->}\ \textit{expr} & \\
\textit{vdecl} & ::= & \text{var}\ =\ \textit{expr} &
\end{array}
$$

EmBounded

HUME Abstract Syntax (cont'd)

exnexpr	::=	int | float | char | bool | string | *	
		| var $expr_1 \cdots expr_n$	$n \geq 0$
		| id $expr_1 \cdots expr_n$	$n \geq 0$
		| con $exnexpr_1 \cdots exnexpr_n$	$n \geq 0$
		| ($exnexpr_1$, ... , $exnexpr_n$)	$n \geq 2$
		| if $exnexpr_1$ then $exnexpr_2$ else $exnexpr_3$	
		| case $exnexpr$ of ⟨ $exnmatch_1$ | ⋯ | $exnmatch_n$ ⟩	$n \geq 1$
		| let ⟨ $exnvdecl_1$, ... , $exnvdecl_n$ ⟩ in $exnexpr$	$n \geq 1$
		| *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*	

expr:	black & green
exnexpr:	black & yellow

Conditional

Annotated Type Rule

$$\frac{\Gamma \;\vdash^{\frac{p}{p'}}_{\frac{m}{m'}}\; e_t : A \mid \Phi \qquad \Gamma \;\vdash^{\frac{p}{p'}}_{\frac{m}{m'}}\; e_f : A \mid \Psi}{\Gamma, x{:}\mathsf{bool} \;\vdash^{\frac{p-1}{p'}}_{\frac{m}{m'}}\; \texttt{if}\ x\ \texttt{then}\ e_t\ \texttt{else}\ e_f : A \mid \Phi \cup \Psi}\ \text{(Conditional)}$$

Annotated Operational Semantics

$$\frac{\eta\Big(\mathcal{V}(x)\Big) = (\mathsf{bool}, \mathtt{tt}) \qquad \mathcal{V}, \eta \;\vdash^{\frac{p+1}{p'}}_{\frac{m}{m'}}\; e_1 \rightsquigarrow \ell', \eta'}{\mathcal{V}, \eta \;\vdash^{\frac{p}{p'}}_{\frac{m}{m'}}\; \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rightsquigarrow \ell', \eta'}$$

(Conditional True)

$$\frac{\eta\Big(\mathcal{V}(x)\Big) = (\mathsf{bool}, \mathtt{ff}) \qquad \mathcal{V}, \eta \;\vdash^{\frac{p+1}{p'}}_{\frac{m}{m'}}\; e_2 \rightsquigarrow \ell', \eta'}{\mathcal{V}, \eta \;\vdash^{\frac{p}{p'}}_{\frac{m}{m'}}\; \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rightsquigarrow \ell', \eta'}$$

(Conditional False)

Reading Rules

$$\frac{}{\mathcal{V}, \eta \; \vdash^{p'+1}_{p'} \vdash^{m}_{m-2} \; e \rightsquigarrow \ell, \eta'}$$

is an abbreviation for

$$\frac{\begin{array}{lll} p = p' + 1 & p \in \mathbb{N} & p' \in \mathbb{N} \\ m = m' + 2 & m \in \mathbb{N} & m' \in \mathbb{N} \end{array}}{\mathcal{V}, \eta \; \vdash^{p}_{p'} \vdash^{m}_{m'} \; e \rightsquigarrow \ell, \eta'}$$

Application

$$\frac{\Sigma^{\mathcal{F}}(\mathit{fid}) = \left(\_;\_; A_1, \ldots, A_a \xrightarrow[p']{p}\xrightarrow[m']{m} C\right) \qquad k \geq 1 \qquad k = a}{y_1{:}A_1, \ldots, y_k{:}A_k \mid \frac{p}{p'+k} \mid \frac{m}{m'} \;\; \mathit{fid}\;\; y_1 \cdots y_k : C \mid \emptyset} \text{(App)}$$

$$\frac{\begin{array}{c}\Sigma^{\mathcal{F}}(\mathit{fid}) = \left(\_;\_; A_1, \ldots, A_a \xrightarrow[q']{q}\xrightarrow[n']{n} C\right) \qquad k \geq 1 \qquad k < a \\ \Phi = \{p = 0, p' = k-1, m = \mathsf{Size}(\mathsf{Closure}_k), m' = 0\}\end{array}}{y_1{:}A_1, \ldots, y_k{:}A_k \mid \frac{p}{p'} \mid \frac{m}{m'} \;\; \mathit{fid}\;\; y_1 \cdots y_k : A_{k+1}, \ldots, A_a \xrightarrow[q']{q+k}\xrightarrow[n']{n} C \mid \Phi}$$

(Under App)

$$\frac{\begin{array}{c}\Sigma^{\mathcal{F}}(\mathit{fid}) = \left(\_;\_; A_1, \ldots, A_a \xrightarrow[q']{q}\xrightarrow[n']{n} C\right) \qquad a \geq 1 \qquad k > a \\ y_1{:}A_1, \ldots, y_a{:}A_a \mid \frac{p}{p'} \mid \frac{m}{m'} \;\; \mathit{fid}\;\; y_1 \cdots y_a : C \mid \Phi \\ \widehat{z} \text{ is fresh} \qquad C = B_1, \ldots, B_b \xrightarrow[q']{q}\xrightarrow[n']{n} E \\ \widehat{z}{:}C, y_{a+1}{:}A_{a+1}, \ldots, y_k{:}A_k \mid \frac{p'}{p''} \mid \frac{m'}{m''} \;\; \widehat{z}\;\; y_{a+1} \cdots y_k : E \mid \Psi\end{array}}{y_1{:}A_1, \ldots, y_k{:}A_k \mid \frac{p}{p''} \mid \frac{m}{m''} \;\; \mathit{fid}\;\; y_1 \cdots y_k : E \mid \Phi \cup \Psi}$$

(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 $\mathsf{Nat} \to \mathsf{List}_n\big(\mathsf{Nat}\big) \to \mathsf{List}_n\big(\mathsf{Nat}\big) \times \mathsf{List}_n\big(\mathsf{Nat}\big)$

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

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.