sicsa*

University of St Andrews

Automatic Amortised Resource Analysis for Hume

Kevin Hammond, Steffen Jost and Hans-Wolfgang Loidl
University of St Andrews, Scotland

http://www.embounded.org

EPSRC SEAS DTC BAE SYSTEMS EmBounded

Hume Research Objectives

University of St Andrews

- **Virtual Testbed for Space/Time/Power Cost Modelling**
  - targetting Embedded Systems
- **Real-Time, Bounded Space High-Level Programming**
  - Based on Combining Functional Programming and Finite Automata
- **Concurrent Multithreaded Design**
  - Asynchronous threading
  - suitable for multicore

In the near future, we will view software without formal resource bounds in the same way as we regard untyped programs today
Greg Morriset, Harvard University

Hume Design Objectives

University of St Andrews

- **Reliability and Predictability**
  - Highly reliable software: *correctness by construction*
  - High degree of determinacy: *scheduling, communication, functionality, behaviour, …*
- **Expressibility and Controllability**
  - High-level language features: *automatic memory management, recursion, …*
  - Low-level interfacing: interrupts, scheduling, fifos, ports, memory-mapped I/O, …
- **Costability**
  - Accurate cost predictions: *space, time, power, concurrency*

What is Resource Analysis?

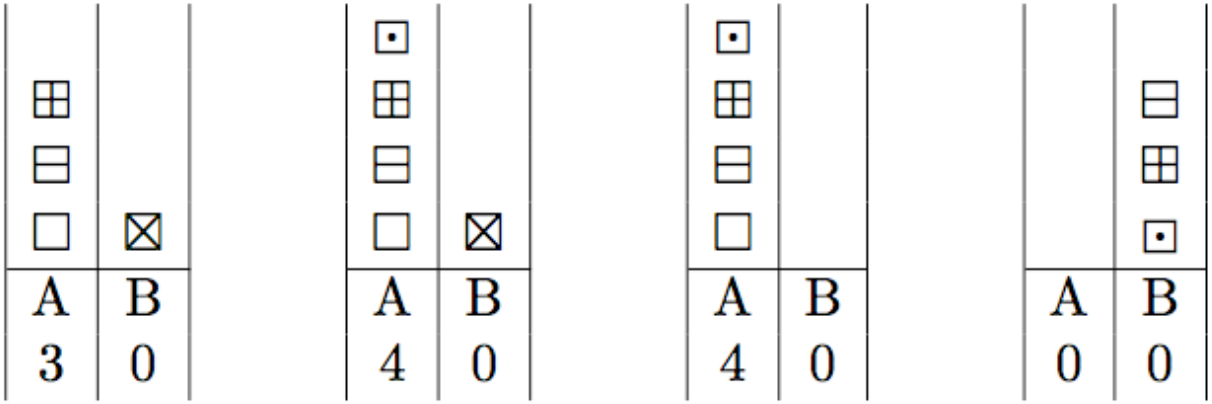
University of St Andrews

- We are trying to determine *a-priori* costs for the use of countable resources
  - time, dynamic memory, stack, power, …
- We need to provide guaranteed bounds on these costs
  - essential to avoid exhausting scarce resources
    - » e.g. memory, file handles, real-time
  - valuable when taking runtime decisions based on resource usage
- We want to give good bounds
  - avoid wasting resources
  - improve usefulness
- We want an efficient analysis
  - analysis should be cheap enough to be part of normal compilation
- We want wide coverage
  - as many programs as possible

Amortisation Example

University of St Andrews

We use *amortisation*. The amortised cost of an operation is its total cost, *plus the difference in potential before/after the operation*



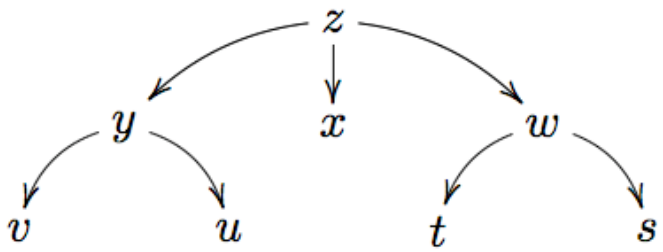
Operation	$\overrightarrow{\texttt{enqu}(\boxdot)}$	$\overrightarrow{\texttt{dequ}() = \boxtimes}$	$\overrightarrow{\texttt{dequ}() = \square}$
Actual Cost	1	1	5
Change of Potential	1	0	−4
Amortised Cost	2	1	1

Example: Simulating a queue by using two stacks, A for enqueue, B for dequeue; if B is empty, content of A is moved to B

Resource costs as types

University of St Andrews

- We can capture costs for each kind of constructor as an annotation. So given a function *mill :: Tree -> Result*, the costs for using *mill* on different trees are as follows

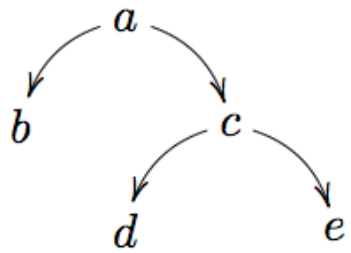


Tree (Node<7> | Leaf <0.5>)

$3 \times 7 + 5 \times 0.5 = \lfloor 23.5 \rfloor$

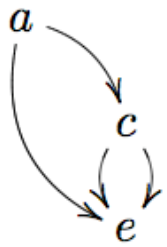
Tree (Node<0.5> | Leaf <7>)

$3 \times 0.5 + 5 \times 7 = \lfloor 36.5 \rfloor$



$2 \times 7 + 3 \times 0.5 = \lfloor 15.5 \rfloor$

$2 \times 0.5 + 3 \times 7 = \lfloor 22 \rfloor$



$2 \times 7 + 3 \times 0.5 = \lfloor 15.5 \rfloor$

$2 \times 0.5 + 3 \times 7 = \lfloor 22 \rfloor$

Costs and Potential

University of St Andrews

- The costs for *mill* represent the *potential* for the state
  - when analysing mill, we only need to keep track of this number
- The potential does not account for sharing
  - this is sensible, since we may need to apply an operation for each node, shared or not!
- The potential can never be negative
  - establishes an upper bound on execution cost

Automating Amortised Analysis

University of St Andrews

- **Type-based approach**
  - first translate into cost-equivalent intermediate form, Schopenhauer
  - one rule per construct
  - plus *substructural* rules for weakening, subtyping etc.
  - potential *threaded* through execution path
  - constraints exposed on potential variables
  - constraints then solved using a standard linear solver (e.g. *lp-solve*)

Schopenhauer

University of St Andrews

- **Intermediate form, simplified version of Hume for analysis**
  - let-normal form - arguments lifted into let-bound variables
  - two forms of let: LET is cost-neutral
  - functions take a fixed number of arguments
  - case! is destructive match
  - lambda-expressions introduced
  - explicit recursion: let rec

$$
\begin{array}{lll}
vars & ::= & \langle\ varid_1\ ,\ \ldots\ ,\ varid_n\ \rangle \qquad n \geq 0 \\
expr & ::= & const \mid varid \mid varid\ \ vars \mid conid\ \ vars \\
 & \mid & \lambda\, varid\, .\, expr \\
 & \mid & \texttt{if}\ varid\ \texttt{then}\ expr_1\ \texttt{else}\ expr_2 \\
 & \mid & \texttt{case}\ \ varid\ \texttt{of}\ conid\ vars\ \texttt{->}\ expr_1\ \texttt{|}\ expr_2 \\
 & \mid & \texttt{case!}\ varid\ \texttt{of}\ conid\ vars\ \texttt{->}\ expr_1\ \texttt{|}\ expr_2 \\
 & \mid & \texttt{let}\ varid\ \texttt{=}\ expr_1\ \texttt{in}\ expr_2 \\
 & \mid & \texttt{LET}\ varid\ \texttt{=}\ expr_1\ \texttt{IN}\ expr_2 \\
 & \mid & \texttt{let rec}\ \left\{ \begin{array}{l} varid_1\ \ \texttt{=}\ expr_1\ \texttt{;} \\ \quad\ldots \\ varid_n\ \ \texttt{=}\ expr_n \end{array} \right\}\ \texttt{in}\ expr \qquad n \geq 1
\end{array}
$$

Operational Semantics Rules

University of St Andrews

- **Each operational semantics rule is of the form:**

$$\mathcal{V}, \mathcal{H} \vdash \frac{n}{n'} e \rightsquigarrow \ell, \mathcal{H}'$$

where

  - **e is the expression to be evaluated**
  - **V is a variable environment mapping variables to values**
  - **H, H' is the heap before after evaluation the expression**
  - **l is the location of the result in the new heap**
  - **n, n' are the associated costs before/after evaluation**

- **So each rule alters the heap in the context of the environment V, producing a result in the new heap, and yielding specific costs**

$$\mathcal{V}, \mathcal{H} \vdash \frac{3}{1} e \rightsquigarrow \ell, \mathcal{H}'$$

3 units are enough to evaluate e

exactly 1 unit is unused after evaluating e

Schopenhauer Operational Semantics (1)

University of St Andrews

$$\frac{n \in \mathbb{Z} \qquad \ell \notin \mathrm{dom}(\mathcal{H})}{\mathcal{V}, \mathcal{H} \vdash^{\frac{q' + \texttt{KmkInt}}{q'}} n \rightsquigarrow \ell, \mathcal{H}[\ell \mapsto (\mathsf{int}, \mathsf{n})]} \text{(OP CONST INT)}$$

$$\frac{w = (\mathsf{bool}, \mathbf{tt}/\mathbf{ff}) \qquad \ell \notin \mathrm{dom}(\mathcal{H})}{\mathcal{V}, \mathcal{H} \vdash^{\frac{q' + \texttt{KmkBool}}{q'}} \texttt{true/false} \rightsquigarrow \ell, \mathcal{H}[\ell \mapsto w]} \text{(OP CONST BOOL)}$$

$$\frac{\mathcal{V}(x) = \ell}{\mathcal{V}, \mathcal{H} \vdash^{\frac{q' + \texttt{KpushVar}}{q'}} x \rightsquigarrow \ell, \mathcal{H}} \text{(OP VAR)}$$

$$\frac{\mathcal{V}^\star = \mathcal{V}\restriction_{\mathrm{FV}(e)\setminus x} \qquad k = |\mathcal{V}^\star| \qquad w = (\lambda x.e\,,\ \mathcal{V}^\star) \qquad \ell \notin \mathrm{dom}(\mathcal{H})}{\mathcal{V}, \mathcal{H} \vdash^{\frac{q' + \texttt{KmkFun}(k)}{q'}} \lambda x.e \rightsquigarrow \ell, \mathcal{H}[\ell \mapsto w]} \text{(OP ABS)}$$

$$\frac{\mathcal{H}(\mathcal{V}(y)) = (\lambda x.e\,,\ \mathcal{V}^\star) \qquad \mathcal{V}^\star\,[x \mapsto \mathcal{V}(x_0)]\,, \mathcal{H} \vdash^{\frac{q - \texttt{Kapp}}{q' + \texttt{Kapp}'}} e \rightsquigarrow \ell, \mathcal{H}'}{\mathcal{V}, \mathcal{H} \vdash^{\frac{q}{q'}} y\ x_0 \rightsquigarrow \ell, \mathcal{H}'} \text{(OP APP)}$$

Schopenhauer Operational Semantics (2)

University of St Andrews

$$\frac{\mathcal{H}(\mathcal{V}(x)) = (\mathsf{bool}, \mathtt{tt}) \quad \mathcal{V}, \mathcal{H} \vdash^{\frac{q - \mathtt{KifT}}{q' + \mathtt{KifT}'}} e_t \rightsquigarrow \ell', \mathcal{H}'}{\mathcal{V}, \mathcal{H} \vdash^{\frac{q}{q'}} \mathtt{if}\ x\ \mathtt{then}\ e_t\ \mathtt{else}\ e_f \rightsquigarrow \ell', \mathcal{H}'} \text{(OP CONDITIONAL TRUE)}$$

$$\frac{\mathcal{H}(\mathcal{V}(x)) = (\mathsf{bool}, \mathtt{ff}) \quad \mathcal{V}, \mathcal{H} \vdash^{\frac{q - \mathtt{KifF}}{q' + \mathtt{KifF}'}} e_f \rightsquigarrow \ell', \mathcal{H}'}{\mathcal{V}, \mathcal{H} \vdash^{\frac{q}{q'}} \mathtt{if}\ x\ \mathtt{then}\ e_t\ \mathtt{else}\ e_f \rightsquigarrow \ell', \mathcal{H}'} \text{(OP CONDITIONAL FALSE)}$$

$$\frac{k \geq 0 \quad c \in \mathsf{Constrs} \quad \ell \notin \mathrm{dom}(\mathcal{H}_k) \quad w = (\mathsf{constr}_c, \mathcal{V}(x_1), \ldots, \mathcal{V}(x_k))}{\mathcal{V}, \mathcal{H} \vdash^{\frac{q' + \mathtt{KCons}(c)}{q'}} c\,\langle x_1, \ldots, x_k \rangle \rightsquigarrow \ell, \mathcal{H}[\ell \mapsto w]} \text{(OP CONSTRUCTOR)}$$

Schopenhauer Operational Semantics (3)

University of St Andrews

$$\frac{\mathcal{H}(\mathcal{V}(x)) \neq (\mathsf{constr}_c, k_1, \ldots, k_k) \quad \mathcal{V}, \mathcal{H} \vdash \frac{q - \mathtt{KCaseF}(c)}{q' + \mathtt{KCaseF}'(c)}\ e_2 \rightsquigarrow \ell, \mathcal{H}'}{\mathcal{V}, \mathcal{H} \vdash \frac{q}{q'}\ \mathsf{case}\ x\ \mathsf{of}\ c\ \langle y_1, \ldots, y_k \rangle\ \texttt{->}\ e_1 \,|\, e_2 \rightsquigarrow \ell, \mathcal{H}'}\ \text{(OP CASE FAIL)}$$

$$\frac{\mathcal{H}(k) = (\mathsf{constr}_c, k_1, \ldots, k_k) \quad \mathcal{V}[y_1 \mapsto k_1, \ldots, y_k \mapsto k_k], \mathcal{H} \vdash \frac{q - \mathtt{KCaseT}(c)}{q' + \mathtt{KCaseT}'(c)}\ e_1 \rightsquigarrow \ell, \mathcal{H}'}{\mathcal{V}[x \mapsto k], \mathcal{H} \vdash \frac{q}{q'}\ \mathsf{case}\ x\ \mathsf{of}\ c\ \langle y_1, \ldots, y_k \rangle\ \texttt{->}\ e_1 \,|\, e_2 \rightsquigarrow \ell, \mathcal{H}'}\ \text{(OP CASE SUCCEED)}$$

$$\frac{\begin{array}{c}\mathcal{H}(k) = (\mathsf{constr}_c, k_1, \ldots, k_k) \\ \mathcal{V}^\star = \mathcal{V}[k \mapsto \mathbf{Bad}, y_1 \mapsto k_1, \ldots, y_k \mapsto k_k] \\ \mathcal{V}^\star, \mathcal{H} \vdash \frac{q - \mathtt{KCaseT}(c) + \mathtt{KCons}(c)}{q' + \mathtt{KCaseT}'(c)}\ e_1 \rightsquigarrow \ell, \mathcal{H}'\end{array}}{\mathcal{V}[x \mapsto k], \mathcal{H} \vdash \frac{q}{q'}\ \mathsf{case!}\ x\ \mathsf{of}\ c\ \langle y_1, \ldots, y_k \rangle\ \texttt{->}\ e_1 \,|\, e_2 \rightsquigarrow \ell, \mathcal{H}'}\ \text{(OP CASE! SUCCEED)}$$

Schopenhauer Operational Semantics (4)

University of St Andrews

$$\frac{\mathcal{V}, \mathcal{H} \vdash^{\frac{q_1 - \texttt{KLet1}}{q_2}} e_1 \leadsto \ell_1, \mathcal{H}_1 \qquad \mathcal{V}_1 = \mathcal{V}[x \mapsto \ell_1] \qquad \mathcal{V}_1, \mathcal{H}_1 \vdash^{\frac{q_2 - \texttt{KLet2}}{q' + \texttt{KLet3}}} e_2 \leadsto \ell_2, \mathcal{H}_2}{\mathcal{V}, \mathcal{H} \vdash^{\frac{q_1}{q'}} \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \leadsto \ell_2, \mathcal{H}_2} \text{(OP LET)}$$

$$\frac{\mathcal{V}^\star = \mathcal{V}[x_1 \mapsto \ell_1, \ldots, x_n \mapsto \ell_n] \qquad \mathcal{V}^\star, \mathcal{H}_n \vdash^{\frac{q_{n+1} - \texttt{KRec3}}{q' + \texttt{KRec4}}} e \leadsto \ell, \mathcal{H} \qquad \forall i \in \{1, \ldots, n\}. \mathcal{V}^\star, \mathcal{H}_{i-1} \vdash^{\frac{q_i - \texttt{KRec2}}{q_{i+1}}} e_i \leadsto \ell_i, \mathcal{H}_i}{\mathcal{V}, \mathcal{H}_0 \vdash^{\frac{q_1 + \texttt{KRec1}}{q'}} \texttt{let rec}\ \{x_1 = e_1; \ldots; x_n = e_n\}\ \texttt{in}\ e \leadsto \ell, \mathcal{H}'} \text{(OP REC)}$$

Amortised Analysis Type Rules

University of St Andrews

- Let Γ be a typing context mapping identifiers to annotated Schopenhauer types. Then

$$\Gamma \vdash^{q}_{q'} e : A \mid \phi$$

means that for all valuations v that satisfy the constraints in Φ, expression e has type v(A) under context v(Γ). Evaluating e in environment V requires a potential of at most v(q) + Φ^{v}_{H}(V: Γ) and leaves a potential of at least v(q) + Φ^{v}_{H}(V: Γ) available.

Valid Schopenhauer Types

University of St Andrews

$$
\begin{array}{rll}
T ::= & \mathsf{int} \mid \mathsf{bool} \mid \alpha & \\
\mid & \mu\alpha.\{\ c_1{:}(q_1, \overrightarrow{T_1}) \mid \ldots \mid c_k{:}(q_k, \overrightarrow{T_k})\ \} & \\
\mid & \forall \vec{r} \in \psi.\overrightarrow{T} \xrightarrow[p']{p} T' & \\
\mid & \forall \vec{\alpha}{:}\psi.T &
\end{array}
$$

type variable

data type:
ci are constructors,
qi are potentials for those constructors

quantified (polymporphic) type

function type:
r are resource variables
ψ are constraints on the rs
p/p' are potential captured by the function

Type Rules: Basic Expressions

University of St Andrews

$$\frac{}{x{:}A \vdash^{\frac{\texttt{KpushVar}}{0}} x : A \mid \emptyset} \text{(VAR)}$$

$$\frac{n \in \mathbb{Z}}{\varnothing \vdash^{\frac{\texttt{KmkInt}}{0}} n : \mathsf{int} \mid \emptyset} \text{(INT)}$$

$$\frac{e \in \{\texttt{true}, \texttt{false}\}}{x{:}A \vdash^{\frac{\texttt{KmkBool}}{0}} e : A \mid \emptyset} \text{(BOOL)}$$

KpushVar etc. are symbolic values for costs in the Hume abstract machine (HAM)
This allows us to vary the costs for different kinds of resources (*resource polymorphism*)
For simple expressions, we always return 0 potential.

Type Rules: LET-bindings

University of St Andrews

$$\frac{\Gamma \vdash^{q_1}_{q_2 + \texttt{KLet2}} e_1 : A_1 \mid \psi_1 \qquad \Delta, x{:}A_1 \vdash^{q_2}_{q_3} e_2 : A_2 \mid \psi_2}{\Gamma, \Delta \vdash^{q_1 + \texttt{KLet1}}_{q_3 - \texttt{KLet3}} \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : A_2 \mid \psi_0 \cup \psi_1 \cup \psi_2}\ (\textsc{Let})$$

The LET rule threads cost through the two expressions e1 and e2.

Type Rules: Functions

University of St Andrews

$$\frac{\begin{array}{c}\mathrm{dom}(\Gamma) = \mathrm{FV}(e) \setminus x \qquad B = A \xrightarrow[q']{q} C \qquad \phi \cup \psi \Rightarrow \xi \\ \Gamma, x{:}A \vdash^{q}_{q'} e : C \mid \xi \qquad \phi \Rightarrow \bigcup_{D \in \mathrm{ran}(\Gamma)} \curlyvee(D \,|\, D, D) \\ \vec{r} \notin \mathrm{FV}_{\diamond}(\Gamma) \cup \mathrm{FV}_{\diamond}(\phi)\end{array}}{\Gamma \vdash^{\mathtt{KmkFun}(|\Gamma|)}_{0} \lambda x.e : \forall \vec{r} \in \psi.B \mid \phi} \text{(ABS)}$$

$\sigma : \vec{r} \to \mathsf{CV}$ a substitution to fresh resource variables

$$\frac{\sigma(B) = A \xrightarrow[q']{q} C}{x{:}A,\ y{:}\forall \vec{r} \in \psi.B \vdash^{q + \mathtt{Kapp}}_{q' - \mathtt{Kapp}'} y\, x : C \mid \sigma(\psi)} \text{(APP)}$$

Potential is carried through the function type. We use a sharing rule in ABS to ensure that we allow for repeated applications of the function.

Type Rules: Algebraic Datatypes

University of St Andrews

$$\frac{c \in \text{Constrs} \qquad C = \mu X.\{\cdots \mid c : (p, \langle B_1, \ldots, B_k \rangle) \mid \cdots\} \qquad A_i = B_i \vee (A_i = C \wedge B_i = X) \text{ (for } i = 1, \ldots, k)}{x_1{:}A_1, \ldots, x_k{:}A_k \vdash^{\frac{p + \texttt{KCons}}{0}} c\,\langle x_1, \ldots, x_k \rangle : C \mid \emptyset} \text{(CONSTR)}$$

$$\frac{c \in \text{Constrs} \qquad A = \mu X.\{\cdots \mid c : (p, \langle B_1, \ldots, B_k \rangle) \mid \cdots\} \qquad \Gamma, y_1{:}B_1[A/X], \ldots, y_k{:}B_k[A/X] \vdash^{\frac{q + p - \texttt{KCaseT}(c)}{q' + \texttt{KCaseT}'(c)}} e_1 : C \mid \phi \qquad \Gamma, x{:}A \vdash^{\frac{q - \texttt{KCaseF}(c)}{q' + \texttt{KCaseF}'(c)}} e_2 : C \mid \psi}{\Gamma, x{:}A \vdash^{\frac{q}{q'}} \texttt{case}\ x\ \texttt{of}\ c\,\langle y_1, \ldots y_k \rangle\ \texttt{->}\ e_1 \mid e_2 : C \mid \phi \cup \psi} \text{(CASE)}$$

In building a constructor using CONSTR we acquire potential p, as defined by the type for that constructor. This potential is released in a CASE rule, which must deal with either successful or unsuccessful matches.

Type Rules: Polymorphism

University of St Andrews

- **We use standard Hindley-Milner typing rules for polymorphism**

$$\frac{\vec{\alpha} \notin \mathrm{dom}(\Gamma) \qquad \vec{\alpha} \notin \psi \qquad \Gamma \vdash^{q}_{q'} e : C \mid \psi \cup \phi}{\Gamma \vdash^{q}_{q'} e : \forall \vec{\alpha}{:}\phi.C \mid \psi} \text{(GENERALISE)}$$

$$\frac{\Gamma \vdash^{q}_{q'} e : \forall \vec{\alpha}{:}\xi.C \mid \psi \qquad \phi \Rightarrow \psi \cup \xi\left[\vec{B} \,/\, \vec{\alpha}\right]}{\Gamma \vdash^{q}_{q'} e : C\left[\vec{B} \,/\, \vec{\alpha}\right] \mid \phi} \text{(SPECIALISE)}$$

Substructural Type Rules

University of St Andrews

- **The RELAX rule allows us to increase costs when needed for analysis**

$$\frac{\Gamma \vdash^{p}_{p'} e : A \mid \phi}{\Gamma \vdash^{q}_{q'} e : A \mid \phi \cup \{q \geq p, q - p \geq q' - p'\}} \text{(RELAX)}$$

- **The SHARE rule captures sharing information**

$$\frac{\Gamma, x{:}A_1, y{:}A_2 \vdash^{q}_{q'} e : C \mid \phi}{\Gamma, z{:}A \vdash^{q}_{q'} e[z/x, z/y] : C \mid \phi \cup \curlyvee(A \,|A_1, A_2)} \text{(SHARE)}$$

This allows us to split z into two variables x and y. Each resource variable in A has equal cost to the sum of its counterparts in A1/A2

Soundness Theorem

University of St Andrews

Theorem 1 (Soundness). *Fix a well-typed Schopenhauer program. Let $r \in \mathbb{Q}^+$ be fixed, but arbitrary. If the following statements hold*

$$\Gamma \vdash^{q}_{q'} e{:}A \mid \phi \qquad (1.A)$$

$$\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow \ell, \mathcal{H}' \qquad (1.B)$$

$$\mathcal{H} \vDash_v \mathcal{V}{:}v(\Gamma) \qquad (1.C)$$

$$v : \text{ a valuation satisfying } v(\phi) \qquad (1.D)$$

then for all $m \in \mathbb{N}$ such that

$$m \geq v(q) + \Phi^v_{\mathcal{H}}\left(\mathcal{V} : v(\Gamma)\right) + r \qquad (1.E)$$

there exists $m' \in \mathbb{N}$ satisfying

$$\mathcal{V}, \mathcal{H} \vdash^{m}_{m'} e \rightsquigarrow \ell, \mathcal{H}' \qquad (1.I)$$

$$m' \geq v(q') + \Phi^v_{\mathcal{H}'}\left(\ell{:}v(A)\right) + r \qquad (1.II)$$

$$\mathcal{H}' \vDash_v \ell{:}v(A) \qquad (1.III)$$

List-processing example: reverse append

University of St Andrews

```
type num = int 16;
data num_list = Nil
              | Cons num num_list;

revApp :: num_list -> num_list  -> num_list;
revApp acc Nil = acc;
revApp acc (Cons x xs) = revApp (Cons x acc) xs;

reverse :: num_list  -> num_list;
reverse xs = revApp Nil xs;

expression reverse;
```

Costs for reverse

University of St Andrews

We first convert the Hume program to a Schopenhauer program:
phamc-an -H -r ra16.hume > ra16.art3

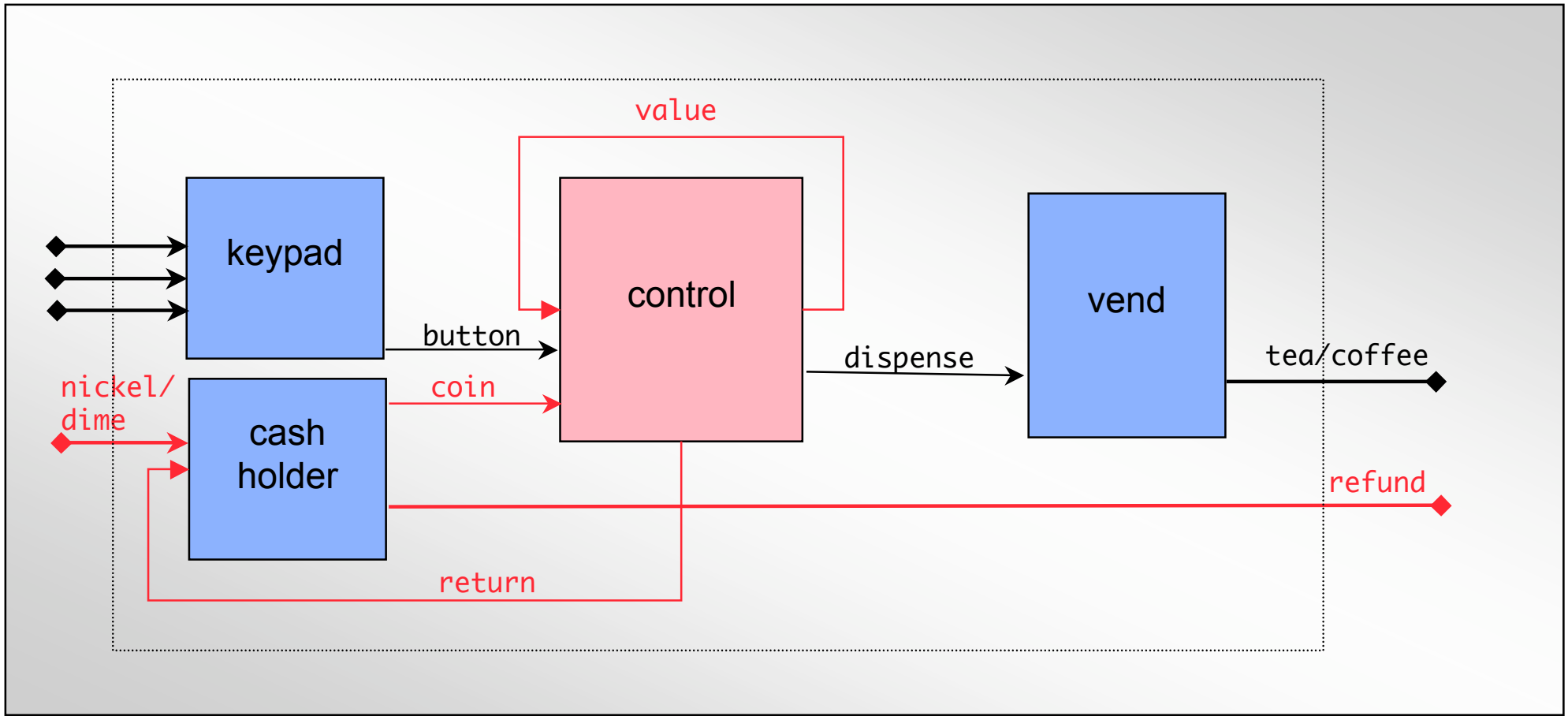
We then analyse the heap consumption:
art3 -H --speak ra16.art3

```
ARTHUR3 typing for HumeHeapBoxed:

0, (num_list[Nil|Cons<4>:int,#]) -(2/0)-> num_list[Nil|Cons:int,#] ,0

Worst-case Heap-units required in relation to input
2 + 4*X1
where       X1 = number of "Cons" nodes at 1. position
```

Total heap consumption is *linear* in the input size: 4 x length input + 2

Example: Vending Machine

University of St Andrews



The control box in Hume

University of St Andrews

```
data Coins =   Nickel   | Dime;
data Drinks =  Coffee   | Tea;
data Buttons = BCoffee | BTea | BCancel;

-- vending machine controller box
box control
in  ( coin :: Coins, button :: Buttons, value :: Int )
out ( dispense :: Drinks, value' :: Int, return :: Int )
match
  ( Nickel, *,       v ) ->  ( *, v + 5, * )
| ( Dime,   *,       v ) ->  ( *, v + 10, * )
| ( *,      BCoffee, v ) ->  vend Coffee 10 v
| ( *,      BTea,    v ) ->  vend Tea 5 v
| ( *,      BCancel, v ) ->  ( *, 0, v );

vend drink cost v = if v >= cost then ( drink, v-cost, * )
                                 else ( *,     v,      * );
```

"*" means ignore input or don't produce output

Costs for the Control Box

University of St Andrews

As before, we first convert the Hume program to a Schopenhauer program:

phamc-an -H -r vending.hume > vending.art3

We then analyse the heap consumption:

art3 -H --shrtcon --speak vending.art3

```
ARTHUR3 typing for HumeHeapBoxed:

@BOX control:
    control.coin:
                wire[W:coins[Nickel|Dime]|NOVAL]
    control.button:
                wire[W<4>:buttons[BCoffee|BTea|BCancel]|NOVAL]
    control.value:
                wire[W<4>:int|NOVAL]
  ---0/0--->
    control.drink:
                wire[W:drinks[Coffee|Tea]|NOVAL]
    control.value':
                wire[W:int|NOVAL]
    control.return:
                wire[W<6>:int|NOVAL]

Worst-case Heap-units required to compute box control once in relation to its input:
  4*X1 + 4*X2
    where
        X1 = one if 2. wire is live, zero if the wire is void
        X2 = one if 3. wire is live, zero if the wire is void
```

NB:
- total heap cost is fixed: doesn't depend on input size;
- return wire has value 6 - reduces bound to 2 from 8

Conclusions

University of St Andrews

- New approach to determining upper bounds on execution costs
  - formally guaranteed bounds (upper and/or lower)
  - "resource polymorphic": heap, stack and execution time
  - uses *amortisation* approach
  - shown here for Hume, but not restricted to pure strict functional languages
    - » extensions to memory cell reuse, assignment, classes, lazy evaluation

Other Approaches

University of St Andrews

- **Sized Types (Vasconcelos and Hammond, IFL 2003)**
  - analyse size of data structures, each type annotated with size
  - costs can then be determined by multiplying sizes by operation costs
  - can be combined with amortised analysis
- **Symbolic Evaluation**
  - convert program to equivalent costed form
  - analysis cost proportional to execution cost (problematic for recursion/iteration)
  - data-dependent, does not give guaranteed bounds
- **Abstract interpretation (Cousot, 1977)**
  - similar to symbolic evaluation, but can resolve fixpoints
    - » cost not proportional to execution cost
  - can find guaranteed bounds, independent of input sizes
  - main difference to type-based approaches is that a specialist analysis engine is used
    - » more powerful, but more complex implementation and proof
- **Profiling**
  - run program and measure execution costs
  - often very high performance overhead
  - costs are usually approximate
  - data-dependent, does not give guaranteed bounds

Ongoing/Future Work

University of St Andrews

- **Lazy Evaluation**
  - new idea of *lazy potential* to suspend payment of potential
- **Non-Linear Bounds/Wider range of applications**
  - combine amortised analysis and sized types
  - investigate *negative* potentials
  - incorporate Campbell's *give-back* annotations for stacks
- **Garbage Collection**
  - Adapt region-based approach to give countable costs
  - Lifetime/Pointer Safety Analysis
    - » An issue if regions are seen as a programmer level notation
    - » *Not really an issue if the mechanisms are to be handled automatically/for experimental testbed purposes*

Some Recent Papers

University of St Andrews

"Carbon Credits" for Resource-Bounded Computations using Amortised Analysis

Steffen Jost, Hans-Wolfgang Loidl, Kevin Hammond, Norman Scaife, and Martin Hofmann

Proc. 2009 Conf. on Formal Methods (FM 2009), Eindhoven, The Netherlands, November 2009.

Worst-Case Execution Time Analysis through Types

Steffen Jost, Hans-Wolfgang Loidl, Norman Scaife, Kevin Hammond, Greg Michaelson, and Martin Hofmann

Proc. 2009 EuroMicro Conf. on Real-Time Systems (Work in Progress Session), Dublin, Ireland, July 2009.

Towards Resource-Certified Software: A Formal Cost Model for Time and its Application to an Image-Processing Example

Armelle Bonenfant, Zezhi Chen, Kevin Hammond, Greg Michaelson, Andy Wallace and Iain Wallace

ACM Symposium on Applied Computing 2007.

A Verified Staged Interpreter is a Verified Compiler: Multi Stage Programming with Dependent Types

Edwin Brady and Kevin Hammond

ACM Conf. on Generative Programming and Component Engineering, October 2006.

Inferring Costs for Recursive, Polymorphic and Higher-Order Functional Programs

Pedro Vasconcelos and Kevin Hammond

Proc. 2003 Intl. Workshop on Implementation of Functional Languages (IFL '03), Edinburgh, Springer-Verlag LNCS, 2004. *Winner of the Peter Landin Prize for best paper*

Predictable Space Behaviour in FSM-Hume,

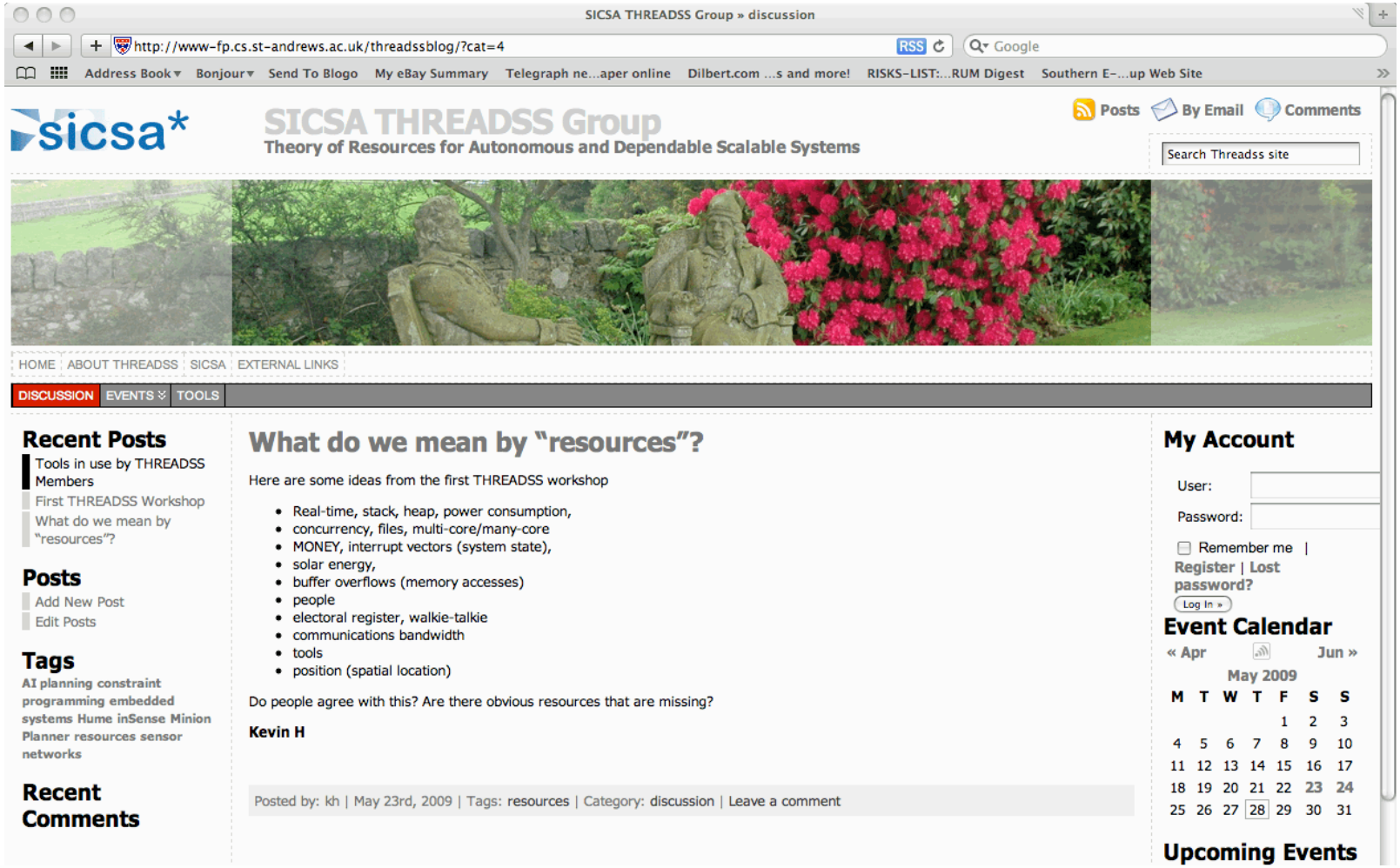
Kevin Hammond and Greg Michaelson,
Proc. 2002 Intl. Workshop on Implementation of Functional Languages (IFL '02), Madrid, Spain, Sept. 2002, Springer-Verlag LNCS 2670, ISBN 3-540-40190-3, 2003, pp. 1-16

http://www.hume-lang.org

University of St Andrews

University of St Andrews

SICSA THREADSS Blog



http://www-fp.cs.st-and.ac.uk/threadssblog